UNIVERSIDADE FEDERAL DE CAMPINA GRANDE

CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA

COORDENAÇÃO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

JOSÉ MANOEL DOS SANTOS FERREIRA

RELATING BUG REPORT FIELDS WITH RESOLUTION STATUS: A CASE STUDY WITH BUGZILLA

CAMPINA GRANDE - PB
2023

Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Relating Bug Report Fields with Resolution Status: A Case Study with Bugzilla

José Manoel dos Santos Ferreira

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Computer Science

Linha de Pesquisa: Software Engineering

Franklin Ramalho and Tiago Massoni

(Supervisor)

Campina Grande, Paraíba, Brasil

©José Manoel dos Santos Ferreira, 05/09/2023

F383r Ferreira, José Manoel dos Santos.
Relating bug report fields with resolution status: a case study with bugzilla / José Manoel dos Santos Ferreira. – Campina Grande, 2023.
93 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2023.
"Orientação: Prof. Dr. Franklin de Souza Ramalho, Prof. Dr. Tiago Lima Massoni".
Referências.

1. Software Engineering. 2. Computer Science. 3. Bug Reports (BR) – Software Quality. I. Ramalho, Franklin de Souza. II. Massoni, Tiago Lima. III. Título.

CDU 004.41(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA SEVERINA SUELI DA SILVA OLIVEIRA CRB-15/225

RELATING BUG REPORT FIELDS WITH RESOLUTION STATUS: A CASE STUDY WITH BUGZILLA

JOSÉ MANOEL DOS SANTOS FERREIRA

DISSERTAÇÃO APROVADA EM 05/09/2023

FRANKLIN DE SOUZA RAMALHO, Dr., UFCG
Orientador(a)

TIAGO LIMA MASSONI, Dr., UFCG
Orientador(a)

EVERTON LEANDRO GALDINO ALVES, Dr., UFCG
Examinador(a)

BRENO ALEXANDRO FERREIRA DE MIRANDA, Dr., UFPE
Examinador(a)

CAMPINA GRANDE - PB

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
POS-GRADUACAO EM CIENCIA DA COMPUTACAO
Rua Aprígio Veloso, 882, Edifício Telmo Silva de Araújo, Bloco CG1, - Bairro Universitário, Campina Grande/PB, CEP 58429-900
Telefone: 2101-1122 - (83) 2101-1123 - (83) 2101-1124
Site: http://computacao.ufcg.edu.br - E-mail: secretaria-copin@computacao.ufcg.edu.br / copin@copin.ufcg.edu.br

FOLHA DE ASSINATURA PARA TESES E DISSERTAÇÕES

JOSÉ MANOEL DOS SANTOS FERREIRA

RELATING BUG REPORT FIELDS WITH RESOLUTION STATUS: A CASE STUDY WITH BUGZILLA

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação como pré-requisito para obtenção do título de Mestre em Ciência da Computação.

Aprovada em: 05/09/2023

Prof. Dr. FRANKLIN DE SOUZA RAMALHO, UFCG, Orientador

Prof. Dr. TIAGO LIMA MASSONI, UFCG, Orientador

Prof. Dr. EVERTON LEANDRO GALDINO ALVES, UFCG, Examinador Interno

Prof. Dr. BRENO ALEXANDRO FERREIRA DE MIRANDA, UFPE, Examinador Externo

Documento assinado eletronicamente por **EVERTON LEANDRO GALDINO ALVES**, **PROFESSOR 3 GRAU**, em 08/09/2023, às 07:34, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **TIAGO LIMA MASSONI**, **COORDENADOR(A) ADMINISTRATIVO(A)**, em 08/09/2023, às 09:08, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **FRANKLIN DE SOUZA RAMALHO**, **PROFESSOR(A) DO MAGISTERIO SUPERIOR**, em 08/09/2023, às 09:18, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

A autenticidade deste documento pode ser conferida no site https://sei.ufcg.edu.br/autenticidade, informando o código verificador **3768514** e o código CRC **0DB99C23**.

Referência: Processo nº 23096.068641/2023-40 SEI nº 3768514

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
POS-GRADUACAO EM CIENCIA DA COMPUTACAO
Rua Aprígio Veloso, 882, Edifício Telmo Silva de Araújo, Bloco CG1, - Bairro Universitário, Campina Grande/PB, CEP 58429-900
Telefone: 2101-1122 - (83) 2101-1123 - (83) 2101-1124
Site: http://computacao.ufcg.edu.br - E-mail: secretaria-copin@computacao.ufcg.edu.br / copin@copin.ufcg.edu.br

REGISTRO DE PRESENÇA E ASSINATURAS

ATA Nº 004/2023 (DISSERTAÇÃO N° 713)

Aos cinco (5) dias do mês de setembro do ano de dois mil e vinte e três (2023), às quatorze horas (14:00), no Auditório do SPLab, da Universidade Federal de Campina Grande - UFCG, nesta cidade, reuniu-se a Comissão Examinadora composta pelos Professores TIAGO LIMA MASSONI, Dr., UFCG, Orientador, funcionando neste ato como Presidente, FRANKLIN DE SOUZA RAMALHO, Dr., UFCG, Orientador, EVERTON LEANDRO GALDINO ALVES, Dr., UFCG, BRENO ALEXANDRO FERREIRA DE MIRANDA, Dr., UFPE, este com participação por videoconferência. Constituída a mencionada Comissão Examinadora pela Portaria Nº 016/2023 do Coordenador do Programa de Pós-Graduação em Ciência da Computação, tendo em vista a deliberação do Colegiado do Curso, tomada em reunião de 14 de Agosto de 2023 e com fundamento no Regulamento Geral dos Cursos de Pós-Graduação da Universidade Federal de Campina Grande - UFCG, juntamente com o Sr(a) JOSÉ MANOEL DOS SANTOS FERREIRA, candidato(a) ao grau de MESTRE em Ciência da Computação, comigo Paloma Nascimento Porto, Secretária(o) dos trabalhos, presentes ainda professores e alunos do referido centro e demais presentes. Abertos os trabalhos, o(a) Senhor(a) Presidente da Comissão Examinadora anunciou que a reunião tinha por finalidade a apresentação e julgamento da dissertação "RELATING BUG REPORT FIELDS WITH RESOLUTION STATUS: A CASE STUDY WITH BUGZILLA", elaborada pelo(a) candidato(a) acima designado, sob a orientação do(s) Professor(es) TIAGO LIMA MASSONI e FRANKLIN DE SOUZA RAMALHO, com o objetivo de atender as exigências do Regulamento Geral dos Cursos de Pós-Graduação da Universidade Federal de Campina Grande - UFCG. A seguir, concedeu a palavra ao (a) candidato(a), o qual, após salientar a importância do assunto desenvolvido, defendeu o conteúdo da dissertação. Concluída a exposição e defesa do candidato, passou cada membro da Comissão Examinadora a arguir o mestrando sobre os vários aspectos que constituíram o campo de estudo tratado na referida dissertação. Terminados os trabalhos de arguição, o(a) Senhor(a) Presidente da Comissão Examinadora determinou a suspensão da sessão pelo tempo necessário ao julgamento da dissertação. Reunidos, em caráter secreto, no mesmo recinto, os membros da Comissão Examinadora passaram à apreciação da dissertação. Reaberta a sessão, o(a) Presidente da Comissão Examinadora anunciou o resultado do julgamento, tendo assim, o candidato obtido o Conceito APROVADO. A seguir, foi encerrada a sessão e lavrada a presente ata, que vai assinada por mim, Paloma Nascimento Porto, pelos membros da Comissão Examinadora e pelo candidato, com exceção do examinador externo BRENO ALEXANDRO FERREIRA DE MIRANDA, o qual receberá cópia da ata e dará ciência e aprovação dos termos, por e-mail. Campina Grande, 5 de Setembro de 2023.

Documento assinado eletronicamente por **EVERTON LEANDRO GALDINO ALVES**, **PROFESSOR 3 GRAU**, em 08/09/2023, às 09:37, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **FRANKLIN DE SOUZA RAMALHO**, **PROFESSOR(A) DO MAGISTERIO SUPERIOR**, em 08/09/2023, às 10:33, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **José Manoel dos Santos Ferreira**, **Usuário Externo**, em 08/09/2023, às 15:39, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **PALOMA NASCIMENTO PORTO**, **ASSISTENTE EM ADMINISTRACAO**, em 11/09/2023, às 10:35, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **TIAGO LIMA MASSONI**, **COORDENADOR(A) ADMINISTRATIVO(A)**, em 12/09/2023, às 17:28, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

A autenticidade deste documento pode ser conferida no site https://sei.ufcg.edu.br/autenticidade, informando o código verificador **3777226** e o código CRC **8A862FB6**.

Referência: Processo nº 23096.068641/2023-40 SEI nº 3777226

Resumo

Os bug reports(BR) são artefatos essenciais para a garantia da qualidade do software. No entanto, o BR produzido, seja por testadores ou usuários, exige do relator uma quantidade considerável de dados, como resumo, etapas necessárias para reproduzir, comportamento esperado/real do sistema, gravidade/prioridade e até mesmo anexos (capturas de tela, vídeos ou arquivos de log). Pesquisas anteriores destacaram a frequência com que esses campos de dados são negligenciados; em resposta, várias diretrizes para escrever bons BR podem ser encontradas na literatura. No entanto, é razoável avaliar o impacto relativo desses campos relatados sobre o resultado dos bugs reportados, especialmente as condições em que eles são resolvidos. Por exemplo, quais campos são os mais importantes para ajudar os desenvolvedores a corrigir um bug? Neste estudo, realizamos uma investigação em um conjunto de dados de 69 mil bugs extraídos da plataforma Bugzilla. Avaliamos cinco modelos de aprendizado de máquina para classificar o status de resolução de bugs (entre FIXED, INVALID, INCOMPLETE, WONTFIX, WORKSFORME, MOVED, DUPLICATED e INACTIVE) e, em seguida, determinamos os recursos que mais influenciam a classificação FIXED. O processo de classificação envolve o emprego de técnicas padrão de aprendizado de máquina para otimização de modelos, incluindo balanceamento, agrupamento e fine-tuning. Notavelmente, o modelo *Random Forest* demonstrou excelente desempenho, alcançando 71,81% de precisão, 74,46% de acurácia e 72,32% de f-measure, com uma notável precisão de 95% na classificação de BR FIXED. Além disso, esse modelo nos permitiu identificar os campos mais influentes para a previsão de resolução. Entre os campos considerados, aqueles relacionados a dados textuais, como resumo, descrição e comentários, surgiram como contribuintes significativos para a classificação de importância do campo. Além disso, os anexos adicionados por meio da seção de comentários mostraram uma relevância considerável para a resolução do BR, assim como as alterações feitas durante o ciclo de vida do BR. Com base nesses resultados, fica evidente que o preenchimento de determinados campos nos BRs pode ajudar na correção dos bugs relatados. Consequentemente, as equipes de desenvolvimento podem se beneficiar dessas descobertas para estabelecer prioridades durante o processo de correção de bugs e alocar recursos de forma mais eficaz para a garantia de qualidade. Além disso, comunicar a importância desses campos aos usuários antes de enviar os BRs pode

resultar em envios mais focados e informativos, além de ajudar a aproveitar melhor o tempo deles.

Abstract

Bug reports are critical artifacts in software quality assurance. However, bug reporting, whether by testers or users, is costly; it demands from the reporter a considerable amount of data, such as summary, steps required to reproduce, expected/actual system behavior, severity/priority, and even attachments (screenshots, videos, or log files). Previous research has highlighted how often these data fields are neglected; in response, several guidelines for writing good reports can be found in the literature. Nevertheless, it is reasonable to assess the relative impact of those reported fields on the outcome of the reported bugs, especially the conditions under which they get resolved. As an inquiry, which fields are the most important for helping developers fix a bug? This study investigates a 69k-bugs dataset extracted from the Bugzilla platform. We evaluate five machine learning models to classify the bug resolution status (among FIXED, INVALID, INCOMPLETE, WONTFIX, WORKSFORME, MOVED, DUPLICATED, and INACTIVE), then determine the features that influence the FIXED classification most. The classification process employs standard ML techniques for model optimization, including balancing, grouping, and fine-tuning. Notably, the *Random Forest* model demonstrated outstanding performance, achieving 71.81% precision, 74.46% accuracy, and 72.32% f-measure, with a remarkable 95% accuracy in classifying FIXED reports. Additionally, this model allowed us to identify the most influential fields for resolution prediction. Among the fields considered, those related to textual data, such as summary, description, and comments, emerged as significant contributors to the field's importance ranking. Furthermore, attachments added through the comments section showed considerable relevance to bug report resolution, as did the changes made throughout the bug report's lifecycle. Given these results, filling specific fields in the bug reports can significantly assist in fixing the reported bugs. Consequently, development teams may benefit from considering these findings to establish priorities during the bug-fixing process and allocate resources more effectively for quality assurance. Moreover, communicating the importance of these fields to reporters before submitting bug reports can lead to more focused and informative submissions and help to make better use of their time.

Agradecimentos

A trajetória até este ponto foi verdadeiramente gratificante, repleta de desafios que foram vencidos, resultando na conclusão deste trabalho. Recordo as palavras de minha estimada mãe, que frequentemente falava o seguinte ditado "Às vezes, fazemos um planos e vira um planeta", e, para alguém que só conhecia uma enxada, hoje posso me orgulhar do título de mestre. Minha história começa na roça, no sertão de Pernambuco, em uma família humilde, com 14 irmãos, e muito trabalho. Aos 15 anos, perdi meu pai, e juntamente com uma de minhas irmãs, parti para trabalhar, um período que me forjou como homem.

Meu pai tinha o desejo de ver seus filhos trilhando o caminho da educação, e, infelizmente, acabei sendo o único a ingressar em uma universidade. Meu sonho era claro: conquistar um lugar em uma instituição de ensino superior. No entanto, esse sonho não apenas se tornou realidade, como ultrapassou todas as expectativas, levando-me a uma jornada que incluiu a graduação, um intercêmbio na Alemanha, e finalmente na minha conquista do título de mestre.

Conforme mencionei inicialmente, essa jornada foi marcada por desafios, mas também por momentos de felicidade e realização. Muitas pessoas desempenharam um papel crucial em meu sucesso, e a todas elas sou profundamente grato. Em primeiro lugar, reconheço que toda a força provém de Deus, a quem acredito estar orquestrando todos os eventos, inclusive minha própria vida. Meus orientadores desempenharam um papel fundamental na conclusão deste trabalho, e expresso minha sincera gratidão pela dedicação e paciência dos professores Franklin e Massoni.

Sou grato ao e-pol por possibilitar a concepção do tema e contribuir significativamente para a minha formação. Minha família, com seu carinho e confiança inabalável em mim, forneceu palavras de encorajamento e motivação essenciais, especialmente nos dias em que a vontade de desistir se fazia presente. Meus amigos sempre estiveram ao meu lado, oferecendo apoio, descontração e alívio nos momentos mais desafiadores. Agradeço a Iann, que sempre esteve presente, fornecendo suporte e discutindo soluções para os obstáculos que surgiram. E também a Lorenna pelo apoio na parte técnica da experimentação.

Meus colegas do Lacina tornaram a jornada mais leve, e o grupo Farofa sempre esteve presente, torcendo por mim. Agradeço a Brito, cujas palavras de motivação foram constantes, a Jackson que sempre feliz me motivava a seguir, assim como a todos os outros que, embora não mencionados individualmente, desempenharam um papel fundamental em minha jornada. A Guilherme e Felipe, que, juntamente comigo, após muito esforço, conseguiram a publicação de dois artigos.

Além disso, minha gratidão se estende a Tiago, meu terapeuta e amigo, que abraçou meu sonho como sua missão, preparando-me para enfrentar os desafios e concluir este trabalho com sucesso. Agradeço de todo o coração a Lilian por seu carinho e ajuda, e a dona Marilene, cujo sorriso radiante e cumprimentos calorosos iluminaram meus dias. Por fim, meu mais sincero agradecimento a todos que fizeram parte desta extraordinária jornada.

Contents

1 Introduction **1**

1.1 Motivation and Context 2

1.2 Problem 6

1.3 Problem Relevance 7

1.4 Objective 8

1.5 Scope 9

1.6 Contribution 9

1.7 Document Organization 10

2 Background **12**

2.1 Bugzilla 12

2.2 Bug Reports 15

2.3 Machine Learning Algorithms 17

3 Research Method **20**

3.1 The ABC framework 21

3.2 Research Questions 22

3.3 Study Subject 23

3.3.1 Data 23

3.3.2 Data collection 24

3.4 Data Balancing 30

3.4.1 Oversampling 31

3.5 Machine Learning Algorithms 32

3.6 Experimental Procedure 33

3.7 Evaluation Metrics 34

4 Results and Discussion 37

4.1 RQ1: What is the best machine learning model for bug report resolution classification? 38

4.1.1 Models' Fine-tuning 39

4.1.2 Grouping Resolutions 46

4.1.3 Cross-Validation 48

4.1.4 Results 49

4.2 RQ2: Which bug report features are more strongly linked to the bug resolution status? 52

4.3 Research Implications 63

4.4 Threats to Validity 64

4.4.1 Construct Validity 64

4.4.2 Internal Validity 65

4.4.3 External Validity 67

4.4.4 Conclusion Validity 68

5 Related Work 69

6 Conclusion 76

6.1 Limitations 76

6.2 Contributions 77

6.3 Future Work 80

List of Symbols

BD - *Balancing & Dumification*

BR - *Bug Report*

D - *Dummification*

EB - *Expected Behavior*

GNB - *Gaussian Naive Bayes*

HIB - *High Impact Bug*

LDA - *Latent Dirichlet Allocation*

ML - *Machine Learning*

NLP - *Natural Language Processing*

N - *Normalization*

NB - *Normalization & Balancing*

NBD - *Normalization & Balancing & Dummification*

ND - *Normalization & Dummification*

OB - *Observable Behavior*

QA - *Quality Assurance*

RNN - *Recurrent Neural Network*

RQ - *Research Question*

S2R - *Steps to Reproduce*

SVM - *Support Vector Machine*

List of Figures

1.1 Bug report 1687674 extracted from Bugzilla. 3
1.2 Fields - Bug report 1663121 extracted from Bugzilla (cropped). 4
1.3 Description - Bug report 1663121 extracted from Bugzilla (cropped). 5

2.1 Bugzilla bug report's life cycle. 14
2.2 Bugzilla bug reporting form. 16

3.1 ABC Framework - Stol and Fitzgerald. 21
3.2 Data Setup before training the machine learning models. 25
3.3 Bug report distribution according to resolution. 31
3.4 Pipeline for machine learning algorithms training, testing, and fine-tuning. . 32

4.1 Random Forest confusion matrix. 45
4.2 Random Forest features importance. 53
4.3 Bug report with a stack trace (cropped). 55
4.4 Boxplot for the Feature *total_words_desc*. 55
4.5 Boxplot for the Feature *total_attachment_comments*. 58

List of Tables

2.1 Main differences between machine learning models. 18

3.1 Bug report features for predicting bug resolution. 26

3.2 Bug report features for predicting bug resolution. 27

3.3 Bug report features for predicting bug resolution. 28

3.4 Bug report resolutions. 29

3.5 Data processing scenarios. 34

4.1 F-Measure by scenario. 39

4.2 Best models metrics results. 39

4.3 Random Forest metrics for individual classes. 40

4.4 Model results after fine-tuning. 44

4.5 Metrics for Random Forest when grouping the resolutions into two classes. 47

4.6 Metrics for Random Forest grouping the resolutions into three classes. . . . 47

4.7 Metrics for Random Forest grouping the resolutions into four classes. . . . 48

4.8 Comparison of classification groupings. 49

4.9 Metrics number of attachments in the comments for bug report description under 40 words. 56

4.10 Descriptive statistics of attachments in the comments for bug report description for all resolutions. 59

4.11 Bug report resolution and comments statistics. 60

4.12 Statistics of the number of changes by bug report resolution. 61

Chapter 1

Introduction

The software development process encompasses several stages: planning, designing, coding, testing, and fixing bugs [18]. Bug reporting plays a crucial role in this process, allowing developers to identify and resolve bugs within the software [5]. However, bug reporting can be challenging, mainly when users have limited time to provide the necessary information [25]. In such cases, focusing on the most important details becomes essential to ensure efficient bug resolution and effective triage. Even though there are works on understanding the fields on the bug report and their importance [99; 59; 58; 47; 32], there is still a gap in understanding the relationship between these fields on the bug report's final resolution. The mentioned gap can be related that most reporters have a different awareness which are the features that are most important to achieve a proper bug fix as resolution.

Moreover, the complete compilation of information within a bug report is constituted by various essential fields that diversify their content and structure. For instance, one essential field is the textual description, where users communicate the bug's characteristics, such as steps to reproduce, observed behavior, expected results, and pertinent system information. Alongside this textual element, other fields, namely platform, component, priority, and severity, also form integral constituents of the bug report. Furthermore, the resolution field is essential in finalizing the bug report, encapsulating the final status of the bug resolution, which can assume distinct values like FIXED or INVALID.

Also, bug reporting requires careful management to optimize resource utilization [4; 53; 69; 64; 81]. Therefore, it is crucial to facilitate the bug reporting process by prioritizing the inclusion of essential information that enables bug reproduction and efficient fixes [20].

Users can effectively communicate the necessary details to developers by adapting bug reports to accommodate time limitations. This approach helps in two ways: First, it focuses on the main information on the bug report to minimize the user time filling it, facilitating the triage process, where bugs are prioritized based on their information robustness and the likelihood of being fixed. Clear and concise bug reports enable triage teams to quickly assess the urgency of each bug and allocate development resources accordingly [5]. Second, when a developer is assigned to resolve a bug, a concise and complete bug report ensures they have the critical information needed to reproduce the bug and find a resolution promptly.

Optimizing bug reporting by considering time limitations and focusing on essential information greatly contributes to the efficiency of bug resolution and the overall software development process. By better understanding the most pertinent fields, developers are expected to reproduce and fix bugs effectively, leading to a more streamlined and productive software development lifecycle.

We adopted the ABC framework proposed by Stol and Fitzgerald [86] to guide our research design and methods. The ABC framework provides a holistic classification of eight research strategies based on two dimensions: generalizability and control. It helped us to select the most suitable strategy for our research question and goal, and to communicate our findings more effectively. We conducted a sample study, which collects data from a large and representative sample of actors from a population of interest and measures their behavior.

1.1 Motivation and Context

In Figure 1.1, we present an illustrative bug report example (Bug ID: 1687674[1]) sourced from Bugzilla for the *web-platform-tests* component in Mozilla. The report is titled "Country Search" and pertains to a defect. The user responsible for this report could be a developer contributor to Mozilla or an individual browsing the platform who discovered the bug and reported it. Notably, the triage team assigned the bug a high priority (P1) and high severity (S1) designation.

Overall, the presented report suffers from significant shortcomings in terms of being a well-written bug report. Firstly, the report lacks a precise summary that clearly conveys the

[1]https://bugzilla.mozilla.org/show_bug.cgi?id=1687674

Closed Bug 1687674 Opened 3 years ago Closed 3 years ago

country search

Categories

Product: Testing

Component: web-platform-tests

Version: Default

Platform: x86_64 All

Type: defect

Priority: P1 Severity: S1

Tracking

Status: RESOLVED INVALID

People (Reporter: , Unassigned)

Details

Votes: 0

Bottom ↓ Tags Timeline

Reporter

Description • 3 years ago

User Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.141 Safari/537.36

Steps to reproduce:

web application, search

Actual results:

retuned all countries

Expected results:

return requested country

Figure 1.1: Bug report 1687674 extracted from Bugzilla.

bug at hand. The current "country search" summary fails to indicate that searching for a country does not yield the expected results. Additionally, there is an absence of any detailed description of the problem. Providing more information about the bug, such as an overview or specific details regarding additional builds and platforms affected, would be beneficial. Lastly, the steps to reproduce the bug are not structured or intuitive enough to ensure accurate replication. The report lacks a description of the interaction with Firefox, making it unclear whether the bug is easily reproducible or occurs sporadically.

In contrast to this problem, numerous studies have extensively discussed the significance of key fields [59; 58] and comprehensive information in bug reports, as they greatly assist developers in identifying and ultimately resolving bugs. However, it is essential to understand what makes an excellent bug report [99]: How much detail should be included, and how can one effectively manage limited time when reporting a bug? In Figures 1.2 and 1.3, we present another example of bug report (1663121[2]) from Bugzilla, specifically addressing

[2]https://bugzilla.mozilla.org/show_bug.cgi?id=1663121

the Printing component in Mozilla. The triage team has classified this bug as a high-priority (P1) defect with significant severity (S2).

Closed Bug 1663121 Opened 3 years ago Closed 3 years ago

Canceling the "Save as" modal while printing to file (Save to Pdf, Microsoft Print to PDF & Microsoft XPS document destinations) locks the new UI

Categories

Product: Toolkit
Component: Printing
Type: defect
Priority: P1 Severity: --

Tracking

Status: VERIFIED FIXED
Milestone: 82 Branch
Tracking Flags:

	Tracking	Status
firefox-esr68	---	unaffected
firefox-esr78	---	unaffected
firefox80	---	unaffected
firefox81	---	disabled
firefox82	---	verified
firefox83	---	verified

People (Reporter: , Assigned:)

References

Blocks: 1658287, 133787
Dependency tree / graph
Regressed by: 1659624
See Also: 1663124

Details

Keywords: regression
Whiteboard: [print2020_v81] [old-ui-]
Has STR: yes
Votes: 0
Bug Flags: mberlinger in-qa-testsuite+

Attachments

PrintToPdf.gif Details
3 years ago , QA
884.41 KB, image/gif

Bug 1663121, reenable print ui form elements if the user cancels saving a pdf Details | Review
3 years ago
47 bytes, text/x-phabricator-request

Figure 1.2: Fields - Bug report 1663121 extracted from Bugzilla (cropped).

The report in Figures 1.2 and 1.3 effectively encompasses the fundamental elements required for an accurate and thorough problem description [22; 99], showcasing the user's dedication to facilitating a prompt and efficient resolution. One notable aspect of the mentioned bug report is the succinct and informative summary, which concisely captures the essence of the encountered bug. By encapsulating the problem within a concise statement, the user enables developers to swiftly grasp the nature of the bug without the need for extensive reading or interpretation.

Furthermore, the Bugzilla user who reported this bug demonstrates an astute understanding of bug reporting conventions by providing essential contextual information [99]. The categorization of the bug under the appropriate product and component ensures that the reporter promptly directs the bug to the relevant development team for evaluation [63;

Affected versions

- 82.0a1 (BuildId:20200903151816)
- 81.0b6 (BuildId:20200903205131)

Affected platforms

- Windows 10 64bit
- Ubuntu 18.04 64bit
- macOS 10.14

Steps to reproduce

1. Launch Firefox.
2. Access any webpage.
3. Hit Ctrl + P
4. Select the Save to Pdf, Microsoft Print to PDF or the Microsoft XPS document destination.
5. Click Print/Save
6. Cancel the Save As window modal.

Expected result

- The UI remains intractable and displayed since the print action was canceled by the user.

Actual result

- The UI options are locked and the user has to close the UI by pressing the esc keyboard button.

Regression Window

- This is a regression.
- Pushlog: https://hg.mozilla.org/integration/autoland/pushloghtml?fromchange=284e3c053bf57ee8f331b21a90751936ee604482&tochange=0de10c26da552953484a56f0fbdb3ab70196988e
- Possible Regressor: ~~Bug 1659624~~

Additional Information

- For further information regarding this issue please observe the attached screencast.
- [Suggested Severity] S2

Figure 1.3: Description - Bug report 1663121 extracted from Bugzilla (cropped).

95]. By adhering to these organizational guidelines, the user effectively aids in expediting the triaging process [96; 55; 97].

In addition to the accurate categorization, the user includes pertinent references to related bugs, regression information, and associated bugs [12; 66]. This thoughtful inclusion enables developers to establish valuable connections and dependencies among various reported problems, streamlining the investigation process and fostering a holistic approach to bug resolution. The meticulous documentation of affected versions and platforms significantly narrows down the scope of the bug, allowing developers to focus their efforts and resources more efficiently. This attention to detail exhibits the user's commitment to providing precise information, ultimately contributing to a more streamlined debugging and resolution process.

The steps to reproduce instructions that the user provides serve as a vital resource for developers in charge of fixing the bug [58; 99; 21]. By meticulously outlining the actions re-

quired to replicate the bug, the user empowers the development team to recreate it accurately, enabling them to observe the undesired behavior firsthand and devise targeted solutions. Additionally, the user's description of the expected and actual results clearly and concisely articulates the bug's impact on the system. This contrast allows developers to ascertain the bug's severity, prioritize their efforts accordingly, and promptly address the discrepancy between expected and observed behavior. Furthermore, the user's inclusion of supplementary materials, such as attachments and additional information, enriches the bug report and enhances its overall quality [47; 45]. The attachments, such as the provided screencast, provide developers with valuable visual insights, further aiding comprehension and expediting debugging.

Given the inherent time constraints of software development, the user's ability to strike a harmonious balance between comprehensive detail and concise presentation is commendable. The user's commitment to providing rich information ensures that developers can effectively address the bug with the utmost efficiency, thereby minimizing unnecessary delays and optimizing the allocation of resources. While it is true that comprehensive bug reports like the one presented are rare, it is important to acknowledge the user's exceptional effort in providing such rich and detailed information. In many software development contexts, users may indeed lack the necessary time, expertise, or resources to produce such comprehensive bug reports [99].

1.2 Problem

For a long time, the term "bug" has been used to refer to software issues, and the way developers and end users report these bugs has become increasingly important. We investigate the problem of reporters does not properly know which field in the bug report has the most importance for the final resolution, that must be filled while reporting a bug. Thus, we focus not on the bug itself but on understanding the fields' impact in the final bug report resolution. Researchers have extensively studied the fields composing a bug report and highlighted their importance. For instance, the steps to reproduce a bug have been widely discussed. Their significance in resolving the bug is highly considered, as well as observed behavior, expected result, concise summary, and description [99; 21; 29; 11;

47]. Studies related to bug report fields are still being conducted nowadays. Hence, our work shows the importance of textual fields on the bug report's final resolution and showcases that summary, description, and comments are the source of essential details, as well as the attachments and the changes made by the reporter through the bug report life. Thus, it complements the information and better illustrates the reported problem, leading the developers to understand the bug better and, consequently, to its final resolution. However, we understand that a major problem is the reporter's knowledge gap in understanding how efficiently and adequately fill out the BRs' fields.

1.3 Problem Relevance

Low-quality bug reports are a significant problem in software development projects that can result in inefficiencies and resource wastage [90; 13; 99]. Since fixing bugs can be a time-consuming and expensive task, having high-quality bug reports is crucial to provide developers with enough information to locate and reproduce the bug [13; 99; 35]. While writing a comprehensive bug report may require additional time and effort, the benefits of a complete report may outweigh the drawbacks. However, when there is no additional time, the available resource must be efficiently used, so focusing on the essential information could contribute to having a complete bug report [59; 58; 99]. Submitting a well-documented bug report increases the likelihood of quickly identifying and resolving the bug [84], compared to submitting a poor-quality report requiring time-consuming triage to determine its validity [90; 13].

For the reason mentioned, providing the right information to locate and fix the bug report becomes valuable knowledge and must be appropriately investigated. The users may be unlikely to consistently provide such complete reports as illustrated in Figures 1.2 ad 1.3. However, with the knowledge about the relationship between BRs' fields and resolution, the reporters can focus their effort and limited time to properly provide these fields that most contribute to FIXED resolution. Additionally, investigating bug reports that have been resolved with resolutions other than FIXED may provide valuable insights to reporters, helping them understand what information is most helpful for developers to identify and address bugs effectively.

Overall, managing and triaging bug reports is crucial for software development projects, and addressing the problem of low-quality bug reports can lead to significant improvements in the efficiency of bug resolution. Developers and users can work together to improve the quality of bug reports by assisting and automating the identification and filtering of invalid bug reports [13]. Providing users with clear instructions on writing quality bug reports may help reduce the number of poor-quality reports, resulting in more accurate and complete bug reports. Additionally, by automating the identification and filtering of invalid bug reports, developers can save valuable time and effort in managing bug reports and instead focus on addressing valid bug reports [88; 94; 35; 88; 45].

Low-quality bug reports are an important issue that deserves attention. It can lead to inefficiencies in software development processes, hinder effective communication between developers and users, and cause resource wastage. Addressing this problem can lead to more effective bug resolution, improved user experiences, and increased efficiency in software development processes.

1.4 Objective

This study aims to investigate bug report resolutions and provide valuable insights into the essential information required in bug reports, particularly the fields that have the most significant influence on achieving the FIXED resolution. Our research includes a comprehensive case study on Bugzilla, as it is one of the most widely used bug-tracking systems [68; 51] and offers detailed information about bugs, users, projects, and collaboration networks. Besides, we use machine learning algorithms (ML) to gain insight into these fields and develop an analysis comparing different algorithms and trying to understand their behavior. Overall, in this research, we seek to better understand the field's impact on the final bug report resolution according to the best machine-learning model and minimize the knowledge gap of the reporters when filling out the bug report. The results form the best model guide for understanding the field's importance and are used to guide the reporter according to its importance. For that, we intend to provide the following:

1. a study in identifying the most important fields in a bug report, according to the machine learning model, that users must effectively provide in the minimal time available;

2. a comparative analyses of five machine learning models on classifying bug report resolution within INVALID, INACTIVE, INCOMPLETE, WORKSFORME, DUPLICATED, WONTFIX, MOVED, and FIXED, from Bugzilla;

3. an investigation on how to improve the best model by applying techniques such as fine-tuning and data preprocessing scenarios;

4. an analysis of the best model results within Random Forest, Logistic Regression, Gaussian Naive Bayes, Decision Tree, and Gradient Boosting in understanding the most influential features and their relation with the FIXED resolution.

1.5 Scope

The scope of this research is the field of Software Engineering in the context of bug reporting and software maintenance. Specifically, this research studies the relationship between bug report fields and the final resolution by applying and analyzing ML algorithms: Random Forest, Decision Tree, Gaussian, Naive Bayes, Logistic Regression, and Gradient Boosting. For that purpose, we use a database (69k bug reports approx.) collected from Mozilla's Bugzilla [2], whose type is defect, where there are several products such as Mozilla, Bugzilla [16], Firefox [36], Thunderbird [38], Toolkit [37] and others.

1.6 Contribution

The study aims to make several significant contributions to the software development process by addressing the knowledge gap between bug report fields and the bug final resolution. We expect that the contributions include improvements in the quality and quantity of information provided in bug reports, as well as more efficient and effective triaging and resolution of bugs. Additionally, the proposed study may provide insight into the gap between the user's and developer's perceptions of what constitutes essential information in bug reports. Thus, this work provides the following contributions:

1. **A study on five machine learning algorithms on bug report resolution classification**: We performed a detailed study on five specific machine learning algorithms

commonly used for bug report resolution classification. We also used appropriate evaluation techniques to compare their performance, including accuracy, precision, recall, and F1 score. Furthermore, we have also analyzed the strengths and weaknesses of each algorithm in the context of bug report resolution classification and identified the most effective algorithm for this task.

2. **Discussion and analysis on the fields more related to bug report FIXED resolution**: We conducted a thorough discussion and analysis of the features that strongly correlate with bug report resolutions, identifying the specific fields or information provided in bug reports that significantly contribute to achieving a FIXED resolution.

This research focuses on bug reporting and software maintenance in the field of Software Engineering. By studying bug report resolutions, particularly the FIXED resolution, and the influential fields/information, we made valuable contributions to improve the software development process. Through the utilization of a large Bugzilla database and machine learning models, this study aims to have implications by enhancing bug report quality, reducing the bug backlog, improving communication between users and developers, and optimizing the use of development resources. However, we have not evaluated the developer's perception in order to attest to whether the fields shown as important by the model are also considered important by them. The main contribution is to support reporters in understanding the vital information they must provide while reporting a bug and the most efficient use of their limited time. Together with a study in the area proportionating discussion and bringing an analysis in the bug reporting data to share the achieved knowledge and insights from this work.

1.7 Document Organization

This document is organized into chapters to provide a clear and structured study presentation. Following the introduction, Chapter 2 refers to the background providing an overview of Bugzilla, the bug reporting system used for this study, as well as the concept of bug reports, data balancing techniques, and the machine learning algorithms employed in the research. Chapter 3 refers to the research method describing the study subject, including details on

data collection and the machine learning algorithms utilized. It also presents the research questions, experimental procedure, and evaluation metrics employed to assess the effectiveness of the models. The results and discussion are in Chapter 4, presenting the study findings and addressing the research questions. It includes an analysis of the best machine learning model for bug report resolution classification, the fine-tuning process, grouping resolutions, identifying bug report features strongly linked to resolution status, and identifying threats to validity. Chapter 5 presents the related work providing a summary of existing literature and research relevant to bug report resolution and machine learning applications in software engineering. Finally, we discuss the conclusion in Chapter 6, highlighting the study's limitations and its contributions to the field of software engineering. It also suggests avenues for future work and potential areas of improvement.

Chapter 2

Background

This chapter provides a comprehensive overview of the concepts of Bugzilla, bug report and bug reporting, machine learning algorithms used, and how they are applied in the software engineering context to improve the quality and efficiency of bug reporting and resolution.

2.1 Bugzilla

Bugzilla is a widely used open-source bug-tracking and issue-tracking system popular among software developers and project managers [68]. It was created in 1998 by Terry Weissman as an internal tool for Netscape Communications Corporation. Still, it was later released as an open-source project under the Mozilla Public License [16].

Bugzilla provides a web-based interface for submitting, tracking, and managing bugs and other software issues. It allows developers and other stakeholders to report, track, and resolve software defects, feature requests, and other issues in a collaborative manner. Bugzilla also offers features such as custom fields [30], email notifications, and advanced search capabilities, making it a versatile and powerful tool for bug tracking in software projects of all sizes. One of the key advantages of Bugzilla is its flexibility and customizability. It can be easily customized to meet the specific needs of different software development teams and organizations. Additionally, Bugzilla integrates well with other software development tools and platforms, such as source code management systems, continuous integration servers, and project management tools.

In Figure 2.1 [15], extracted from the Bugzilla documentation, the comprehensive life cy-

cle of a bug report is depicted, showcasing all possible statuses and potential resolutions as outlined in the Bugzilla manual. The life cycle begins with discovering and reporting a new bug, initially assigned to the UNCONFIRMED state. Subsequently, the developers analyze the bug to determine its validity for further work, and when it is valid, the state becomes NEW. If confirmed, the state is transitioned to CONFIRMED during the triage process, where the developer thoroughly reviews the report. However, if the developer determines that it is not a bug, the state is changed to RESOLVED. The report is then subjected to verification by the Quality Assurance (QA) team, leading to the final state of VERIFIED. In order to transition to the RESOLVED state, the bug report must have a resolution selected from a set of predefined options, including FIXED, DUPLICATED, WONTFIX, WORKSFORME, INCOMPLETE, INVALID, MOVED, INACTIVE, SUPPORT, and EXPIRED. Notably, as depicted in Figure 2.1, the state of a bug report can be modified at any point, allowing for transitions to any other state. For example, a bug report that has been RESOLVED can be changed back to UNCONFIRMED.

In the bug triaging and fixing process, the responsibility of the developer assigned to triage is to manually determine whether a bug report is confirmed as a genuine bug. This confirmation necessitates evaluating each newly submitted bug report. However, the sheer volume of bug reports can be overwhelming. For instance, according to Yuanrui et al. [35], the Mozilla project receives an average of 307 new bug reports daily. Additionally, Anvik et al. [3] estimated that in projects like Eclipse and Firefox, approximately 6% and 7% of bug reports, respectively, are ultimately resolved as INVALID. These statistics highlight the substantial number of bug reports that could have been avoided, saving valuable time spent on manually verifying their validity. A fast search on Bugzilla in 2021 reveals around 6500 INCOMPLETE bug reports and 2400 INVALID bug reports. The cumulative total of these numbers signifies a significant amount of time that could have been better utilized.

Many organizations and projects have been adopting Bugzilla, including the Mozilla Foundation [2], Red Hat [76], the GNOME Project [93], and the Eclipse Foundation [39]. Its use has been documented in many research studies and software engineering publications, demonstrating its importance and value in the software development community. Some specific examples of how Bugzilla has been used in different organizations include its use by the Mozilla Foundation to track bugs and feature requests for its various products [1] and

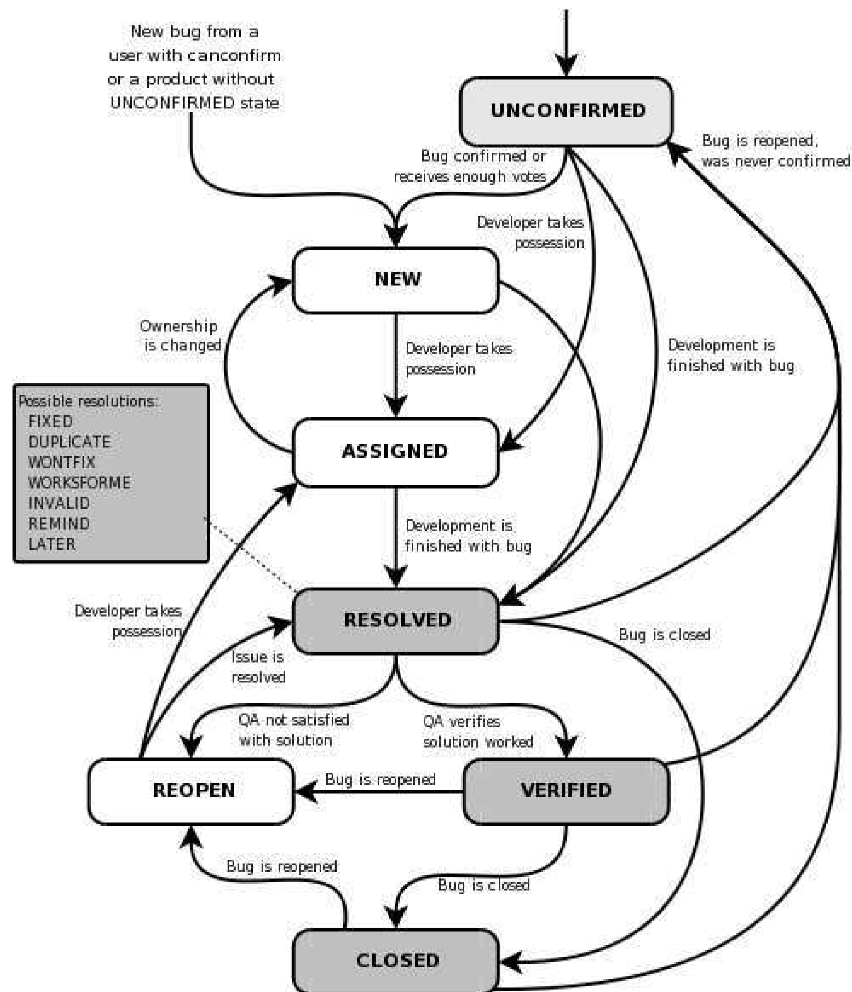


Figure 2.1: Bugzilla bug report's life cycle.

its use by Red Hat to manage issues for its Linux distribution. Despite its many advantages, some limitations and challenges are associated with using Bugzilla. For example, some users may find its interface less intuitive than other bug-tracking systems. However, the Bugzilla community has worked to address these challenges through ongoing development and improvement of the tool.

In comparison to other bug tracking and issue tracking systems such as JIRA [54], Bugzilla offers a similar set of features and capabilities. However, some users may prefer one tool over another based on usability, cost, and integration with other tools. Ultimately, the choice between Bugzilla and other systems will depend on the user's or organization's specific needs and preferences.

Overall, Bugzilla is a well-established and widely used platform for bug management in open-source projects [68]. Its web-based interface allows for easy submission and tracking of

bugs and collaboration between developers and users. Using Bugzilla in this research could provide valuable insights into the bug reporting and fixing process, including the factors contributing to effective bug reporting and resolution.

2.2 Bug Reports

Bug reports come in many different forms, but they typically include information about the software program, the environment in which the bug occurred, and the steps taken to reproduce the problem. The more information the report includes, the easier for the developer to identify and fix the issue [99].

The bug reporting form in Bugzilla consists of various fields that help users accurately describe and report software bugs, as stated in Figure 2.2. The first field is the *Summary*, where users provide a concise title or description of the bug. This field helps in quickly identifying and categorizing the reported bug. The following field is *Product*, which specifies the software or project affected by the bug, ensuring that the bug report reaches the appropriate development team for investigation.

The *Version* field allows users to specify the particular version of the software in which the bug was encountered. This information helps developers determine if the bug has been fixed in subsequent releases or if it is a known bug. The *Component* field allows users to specify the specific module or component of the software where the bug occurred, aiding in identifying the relevant codebase for debugging. The *"What did you do?"* field allows users to describe their steps before encountering the bug. This section should include a detailed sequence of actions developers can follow to reproduce the bug.

The *"What happened?"* field is where users describe the actual results or behavior they observed when encountering the bug. This section should include specific error messages, unexpected outputs, or any other anomalous behavior. Conversely, the *"What should have happened?"* field outlines the expected results or the correct behavior that the user anticipated. This information helps developers understand the user's expectations and provides a clear target for resolving the bug. The *"Attach a File"* field allows users to include any relevant files or screenshots that can aid in understanding and debugging the issue. The *Bug Type* field enables users to categorize the bug based on its severity, priority, or other predefined

Summary: A not working property

Product: Toolkit (Change) Version:

Component: (List) General

For bugs in Toolkit code that don't fit into any of the other components.

What did you do? (steps to reproduce)

What happened? (actual results)

What should have happened? (expected results)

Include your browser's user agent string in the bug description. This can be helpful to developers working on your issue.

Attach a file: Procurar... Nenhum arquivo selecionado.

Bug Type: This is a defect report. This is a request for enhancement.

Security: Many users could be harmed by this security problem: it should be kept hidden from the public until it is resolved.

Submit Bug

Figure 2.2: Bugzilla bug reporting form.

classifications. The *Security* field indicates if the bug report contains sensitive information or relates to a security vulnerability that needs immediate attention and restricted access.

The importance of bug reports cannot be overstated [47]. Without bug reports, developers would have a harder time finding and fixing bugs in their software. Bugs can cause a range of problems, from minor annoyances to critical errors that can cause the software to crash or even become unusable [47]. In addition to helping developers identify and fix bugs, bug reports also provide a valuable source of feedback for software users. By submitting bug

reports, users can help developers improve their software and make it more user-friendly [13]. Bug reports can be submitted by anyone who encounters a bug with a software program. This includes end-users, testers, and developers themselves. Also, bug reports can be submitted through a variety of channels, including email, online forums, and bug-tracking systems like Bugzilla.

Recently, there has been an increasing focus on the quality of bug reports. Researchers have studied the factors that contribute to high-quality bug reports and have developed tools and techniques to help improve the process of bug reporting [99; 47; 21; 22; 97; 13; 84; 70]. These efforts aim to make it easier for developers to identify and fix bugs in their software and improve the overall quality of software products.

2.3 Machine Learning Algorithms

Machine learning is a powerful tool for making predictions based on data. One example of how we can use machine learning to make predictions is in finance, where machine learning algorithms can be trained on historical stock market data to predict future stock prices. The algorithm analyzes patterns and trends in the data and uses this information to make predictions about future stock prices. Also, we have used only supervised machine learning algorithms because Bugzilla provided the desired output (bug report resolution), which we aim to predict. However, selecting appropriate algorithms is a critical step toward achieving accurate and reliable predictions. In this study, we utilized five supervised machine learning algorithms to analyze the bug report dataset: Random Forest [62], Decision Tree [61], Gradient Boosting [17], Logistic Regression [10], and Gaussian Naive Bayes [92]. We selected these algorithms based on their ability to handle the specific characteristics of the bug report dataset, such as the presence of categorical and continuous features, for example, components and number of attachments; class imbalance, as the most common is FIXED and there are few samples from MOVED for instance; and the need for interpretability. We used these five models to provide a comprehensive dataset analysis and identify the most effective approach for predicting bug report resolution. Table 2.1 shows some of the main characteristics and differences within the models.

These five algorithms have been used in academia in the software engineering context and

Table 2.1: Main differences between machine learning models.

Model	**Main Differences**
Random Forest	- Ensemble model that combines multiple decision trees and aggregates their predictions. Reduces overfitting by using random subsets of features for each tree and averaging predictions [83]. Provides a measure of feature importance [78]. Generally performs well with high-dimensional data and handles missing values effectively [52].
Decision Tree	- Builds a single tree model by recursively partitioning the data based on feature thresholds. Also, it allows easy interpretation and visualization. Prone to overfitting, especially with complex or noisy data. However, do not handle missing values well [8; 77].
Gradient Boosting	- Ensemble model that combines multiple weak prediction models sequentially [71]. Minimizes errors by adjusting model weights during training. Can handle a variety of data types. Less prone to overfitting compared to a single decision tree. Requires careful tuning of hyperparameters and may be computationally expensive [75; 17].
Logistic Regression	- Supervised learning algorithm that predicts binary or multilabel outcomes. It models the probability of classes using a logistic function. It also provides interpretable coefficients for each feature and assumes a linear relationship between features and the log-odds of the target variable. Requires feature scaling and may struggle with non-linear relationships [87; 79].
Gaussian Naive Bayes	- Probabilistic model based on Bayes' theorem. It assumes independence between features given the target variable. Also, it follows a Gaussian distribution for continuous features, requires a small amount of training data, and performs well with high-dimensional data. Can be sensitive to correlated features [92; 43].

are applicable for properly using classification bug report resolution. For instance, a study by Kumar et al. [62] used Random Forest to predict software faults in open-source projects, achieving high accuracy and outperforming other machine learning algorithms; Goyal and Sardana [41] used Decision Tree to assess the performance of bug fixing processes in open-source repositories; Cerón-Figueroa et al. [19] used Stochastic Gradient Boosting to predict the maintenance effort of software-intensive systems, achieving high accuracy and demonstrating the usefulness of the model for identifying critical bugs; Tan et al. [89] used Logistic Regression to predict the severity of bug reports in open-source projects, achieving high accuracy and demonstrating the model's usefulness for identifying critical bugs; and Dommati et al. [32] found that using a probabilistic Naive Bayes approach for classifying network bugs was effective.

Chapter 3

Research Method

We performed an exploratory study, aiming to understand the relationship between BR fields and resolution status, and in this way, helping the user invest their limited time into information that is essential to be in the report. In order to achieve this information, we assess different machine learning models' effectiveness in predicting bug report resolution. We investigated two research questions: 1) Which machine learning model performs better in predicting bug resolution status? and 2) Which bug report features are more strongly linked to the bug report resolution status? The findings provide insights into model performance, feature importance, and guidance for improving bug reporting practices.

We performed an experiment in order to understand features or actions that best indicate a bug report resolution and how it influences resolving the bug as FIXED. Once a dataset and a set of previously selected features were available, we used five machine learning algorithms to predict the resolution of the bug report. It was done through a combination of procedures to achieve the best results for the chosen machine learning models.

The rest of this chapter is organized as follows. Section 3.1 presents the ABC framework we used to guide our research strategy and methods. Section 3.2 describes the research questions that motivated our study. Section 3.3 introduces the study subject, including the data sources and collection process. Section 3.4 explains how we balanced the data to avoid class imbalance problems. Section 3.5 presents the machine learning algorithms we applied to build predictive models for bug report resolution. Section 3.6 describes the experimental procedure and Section 3.7 the evaluation metrics that we used to assess the performance of our models.

3.1 The ABC framework

The ABC framework for research strategy is a model that helps researchers design and evaluate their studies. It stands for Actors, Behavior, and Context, which are three aspects of research that need to be balanced. Klaas-Jan Stol and Brian Fitzgerald proposed the framework in their book Contemporary Empirical Methods in Software Engineering [86]. According to the ABC framework, researchers need to consider the following trade-offs when choosing a research strategy: Generalizability over actors (A): how representative are the study participants to the target population?; Precise control of behavior (B): how well can the researcher manipulate and measure the variables of interest?; Realism of context (C): how similar is the setting of the study to the real-world situation?

The ABC framework suggests that these three aspects cannot be maximized simultaneously, and different research strategies have different strengths and weaknesses. The framework also provides examples of eight archetypal research strategies, such as laboratory experiments, formal theory, computer simulation, and experimental simulation [86]. We used the ABC framework to help plan our studies by identifying the most suitable research strategy for our research question and the limitations and challenges we may face.

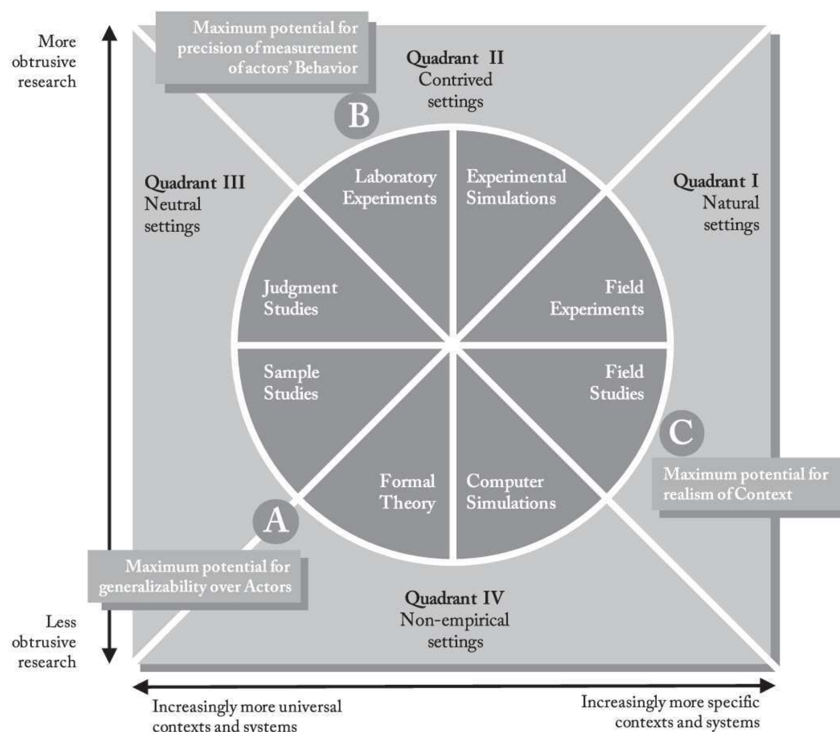


Figure 3.1: ABC Framework - Stol and Fitzgerald.

Our research is a *Sample Study* located at Quadrant III in the Image 3.1. A *sample study* is a research strategy that uses a representative sample of actors from a population of interest and measures their behavior or opinions using surveys or questionnaires. A sample study has high generalizability over actors (A) but low control of behavior (B) and low realism of context (c). A sample study can be used to answer descriptive or explanatory questions about the characteristics or relationships of a population. The following characteristics of the research can justify the archetypal strategy of the sample study:

- The research uses a dataset of bug reports from software projects as a representative sample of a larger population of interest: the software projects. This implies that the research has high generalizability over actors (A), as the results can be applied to the whole population or similar populations.
- The research measures the behavior of the sample using machine learning models and feature analysis. This implies that the research has low control of behavior (B), as the researchers do not manipulate or intervene in the behavior of the sample but only observe and measure it.
- The research performs an experiment in a simulated environment, using a combination of procedures to achieve the best results for the chosen machine learning models. This implies that the research has low realism of context (c), as the experiment does not reflect the real-world context of bug reporting and resolution but only approximates it.

3.2 Research Questions

The primary goal of this study is to delve into the crucial information that a reporter should include in a bug report and understand the relationship between BR and resolution status. We examine the efficacy of five machine learning models in predicting bug report resolutions, specifically focusing on reports resolved as FIXED. Subsequently, we conduct an in-depth analysis of the characteristics associated with these reports. To accomplish this, we address the following research questions:

RQ1. Which machine learning model performs better in predicting bug resolution status? It aims to identify the best ML model for BR resolution prediction. In order to

answer this question, we compared the performance of five different machine learning models, namely Random Forest, Decision Tree, Gradient Boosting, Logistic Regression, and Gaussian Naive Bayes. We evaluated the models using various metrics such as accuracy, precision, recall, and F1 score. The goal is to determine which model performs better in predicting bug resolution status. Additionally, we performed a sensitivity analysis to understand how changing the model's parameters impacts its performance. This analysis helps us understand the selected model's strengths and weaknesses and provide insights for future work.

RQ2. Which bug report features are more strongly linked to the bug resolution status? Aim to identify which bug report features are more strongly linked to the bug resolution status. By analyzing the feature importance of the best classification model, we identified the most important features for prediction according to the best prediction model. This analysis can help developers and testers better understand the characteristics of bug reports that are more likely to be resolved. Additionally, this knowledge can help in improving the quality of bug reports, as it can guide developers, testers, and final users to provide more useful and relevant information in the bug reports.

3.3 Study Subject

To conduct our research, we selected Bugzilla as our study subject, a widely used bug-tracking system known for its diverse projects and open-source nature [27]. The used dataset comprises bug reports from Mozilla's Bugzilla [2], where there are several products. Thus, to collect the bug report data, we developed a Python script that utilized the Bugzilla API [28] to extract relevant fields for our study that will be further detailed.

3.3.1 Data

In this research, we utilize Bugzilla as our chosen study subject and collect a diverse dataset encompassing various products such as Firefox, Bugzilla, Mozilla, Thunderbird, and more. We selected Bugzilla for its wide range of projects and the wealth of information available in bug reports, greatly enriching our research. Moreover, Bugzilla's open-source nature aligns with our objective of transparency and reproducibility. On the Bugzilla platform, the

bug reports are distinguished by the types of enhancement, task, and defect. However, our analysis focuses on defects only, where software behavior deviates from expected norms, including regressions, crashes, and errors [72]. To conduct our experiments, we compiled a dataset of 68,492 bug reports spanning a ten-year period from January 1st, 2013, to January 1st, 2022. We chose this timeframe to ensure a comprehensive data representation, including older and more recent bug reports.

In our comprehensive dataset, we have meticulously curated 65 distinct software projects, each characterized by a dynamic and varied array of bug reports, spanning from as few as one to an astonishing 31,000 reports. This intrinsic variance in the volume of bug reports engenders an inherent high class imbalance, offering a rich and multifaceted dataset that authentically mirrors the intricate landscape of software development. This diversity in bug report frequencies endows our dataset with a broad and realistic spectrum of real-world scenarios, thereby enhancing the efficacy of our machine learning models in the context of software development.

3.3.2 Data collection

We developed a Python script utilizing the Bugzilla API to collect the dataset, which is available on Github[1]. This script facilitated the extraction of bug report information, including primary data, comments, and changes from the Bugzilla platform. Additionally, we developed an auxiliary script to crawl through the downloaded data and extract the relevant features used in our experiments. To better illustrate the processes used in our work, Figure 3.2 presents the steps we took to prepare the data for the study, including the **Feature Cleaning**, **Dummification** and **Balancing**, which we explain ahead.

Initially, we retrieved, from Bugzilla, ten (10) years' worth of public Bug Reports (BRs) belonging to several open-source software products. Our Python script initially fetched almost 68,500 BRs. We used three main endpoints from the Bugzilla API [28] to extract this information, and each one is described below.

1. **Bug Primary Data Endpoint**: The bug primary data endpoint refers to the primary information associated with individual bug reports in Bugzilla. This endpoint provides

[1] https://github.com/manoelf/master-research-bugreport

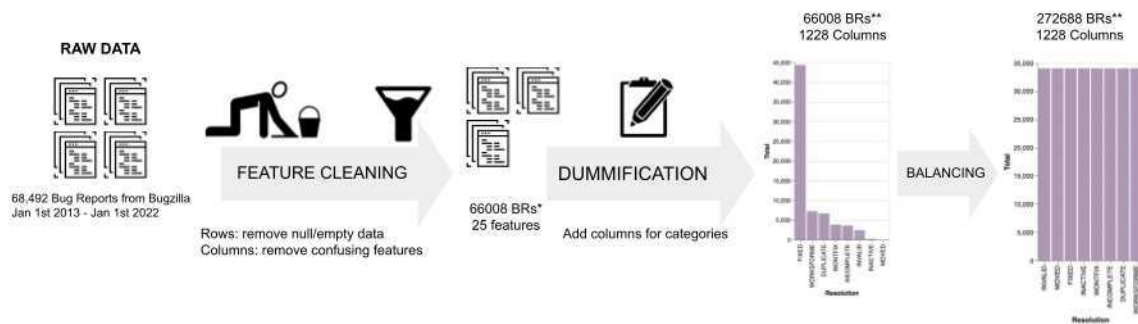


Figure 3.2: Data Setup before training the machine learning models.

essential details such as the bug ID, bug title, bug description, creation date, severity, priority, status, and assigned developer. Bug ID serves as a unique identifier for each bug report, enabling easy referencing and linking with other data sources. The bug title succinctly describes the bug, and the description provides a detailed account of the bug, including steps to reproduce and observed behavior. Other fields, such as creation date, severity, priority, status, and assigned developer, offer valuable insights into the bug's lifecycle and its impact on the software development process. From this endpoint, it is 44 fields available, and we have used 12 of them. The reason is that most were about users involved as cc and QA contact or had many empty values.

2. **Comment Data Endpoint**: The comment data endpoint encompasses the comments and discussions made by various stakeholders during the bug's lifecycle. Each bug report may have multiple comments from developers, testers, users, or project managers. These comments can provide crucial information about bug resolutions, workarounds, testing outcomes, and user feedback. By accessing the comment data endpoint, researchers can gain insights into the collaborative efforts undertaken to identify, resolve, and document bugs. Comment data often includes fields such as comment ID, author, creation date, comment text, and any attachments or links shared within the comments. From this endpoint, it is ten fields available for each comment, from which we have used to extract information such as the number of comments, the number of attachments added to comments, etc.

3. **Changes Data Endpoint**: The changes data endpoint focuses on the historical changes

made to bug reports throughout their lifecycle. When a bug undergoes modifications, such as status updates, reassignments, attachments, or resolution changes, these alterations are recorded as changes. The changes data endpoint provides access to this valuable information, allowing researchers to analyze the evolution of bug reports over time. Examples of fields available through the changes data endpoint include change ID, change date, changed field (e.g., status, resolution), previous value, and new value. This endpoint enables the exploration of patterns, trends, and decision-making processes related to bug resolutions and bug tracking activities. The information extracted from this endpoint is related to the total of changes in the bug report, such as the number of changes done by the reporter, the number of changes in the priority and severity, etc.

After running the script, we extracted 21 features and the response variable (resolution), as shown in Tables 3.1, 3.2 and 3.3.

Table 3.1: Bug report features for predicting bug resolution.

Feature	**Description / Reason for Choosing / Importance**
resolution	The final solution of the bug report, detailed in Table 3.4.
bg_number	Bug report ID. Besides being an ID, it also highlights the timing at which the bug has been reported; as it grows means the bug report is recent, which could introduce a semantic for the machine learning models.
has_attachment	Boolean indicating whether the bug report has an attachment. Attachments can provide additional information or evidence for understanding and resolving the bug.
total_attachment_comments	Number of attachments added in the comments. It indicates the presence of additional information or discussions related to the bug.

Table 3.2: Bug report features for predicting bug resolution.

Feature	**Description / Reason for Choosing / Importance**
severity	Bug severity (blocker, critical, major, normal, minor, trivial, enhancement). The severity provides insight into the bug's impact on the system's functionality.
number_changes_severity	Number of changes in the bug report's severity field. It reflects the changes in the perceived impact or severity of the bug.
priority	Bug report severity (P1, P2, P3, P4, P5). Bug priority indicates the urgency and importance of resolving the bug.
number_changes_priority	Number of changes in the bug report's priority field. It indicates the significance and evolution of the bug's priority over time.
op_sys	The operating system where the bug was found. The operating system may influence the bug's behavior and resolution process.
platform	This is the hardware platform against which the bug was reported (ARM, ARM64, x86, x86_64, etc). The platform can affect the bug's behavior and may require specific platform-related solutions.
component	The component where the bug was found. Different components may have varying levels of complexity and impact on bug resolution.
product	The product where the bug was found. Different products may have different bug resolution processes and priorities.
version	Product version. Different versions of the product may have varying bug resolution priorities and strategies.
votes	Number of votes by Bugzilla users for the bug report. It may reflect the level of community interest and perceived importance of the bug.

Table 3.3: Bug report features for predicting bug resolution.

Feature	**Description / Reason for Choosing / Importance**
total_changes	Number of changes in the bug report across all the fields in its life. The overall activity level on the bug report, including all types of changes
total_users_changes	Number of different users that have made any change in the bug report. It may represent the collaborative effort in resolving the bug and the level of community involvement.
number_changes_assigned	Number of changes in the bug report's assigned developer, expressing how many developers have been assigned to the bug. It helps measure the activity level and collaboration in resolving the bug report.
number_comment	Number of comments on the bug report. More comments could suggest a complex or critical bug that requires additional discussion and attention.
total_comments_by_author	Number of comments added by the bug report author. The author's involvement and engagement may influence the bug's resolution.
total_users_commenting	Number of different users commenting on the bug report. It may indicate the level of discussion and attention the bug has received from multiple stakeholders.
total_words_desc	Number of words in the bug report description. More detailed bug descriptions may provide clearer information for effective resolution.
total_words_summary	Number of words in the bug report summary. The summary provides a concise overview of the bug's nature and can help understand its resolution requirements.

In addition, we removed features directly related to the bug's resolution, which, if maintained, could bias the classification results. The features removed are: *changes_resolution*, indicating the number of changes in the resolution according to its history, which is the model's target label; *total_changes_status*, *status_RESOLVED* and *status_VERIFIED*, using

the status as a feature in a bug report resolution classifier can potentially introduce a bias in the model's understanding of the relationship between the status and resolution - if the model is trained on data where certain statuses are more commonly associated with specific resolutions, it may learn to rely heavily on the status feature to make predictions; also, some other features that had too many empty values, such as *flags*, which are the flags that users can sign a report and comments; *assigned_to* and *creator* were also removed because we do not intend in working with the developer profile as features to the classifier.

As depicted in Figure 3.2, the data setup included the application of Dummification [56] to handle the diverse set of non-numeric features extracted from Bugzilla bug reports. The following non-numeric features were dummified: *has_attachment*, *severity*, *platform*, *priority*, *status*, *version*, *type*, *product*, *component*, and *op_sys*. This process resulted in 1228 columns, which we used as input to train the models.

Table 3.4: Bug report resolutions.

Resolution	Description	Total
MOVED	Used when Bugzilla is not the proper place for the bug reported.	45
INACTIVE	The bug was reported a long time ago, no additional information was given, and there is no way to move forward to fix it.	226
INVALID	The report is not an actual bug.	2,411
INCOMPLETE	There is not enough information to reproduce the bug.	3,624
WONTFIX	The person who triages the bug decides that the bug does not worth fixing.	3,841
DUPLICATE	The bug has been reported more than once.	6,701
WORKSFORME	Attempts to reproduce the bug were not successful, and no bug was found.	7,253
FIXED	The bug reported was fixed and satisfactorily resolved.	44,377

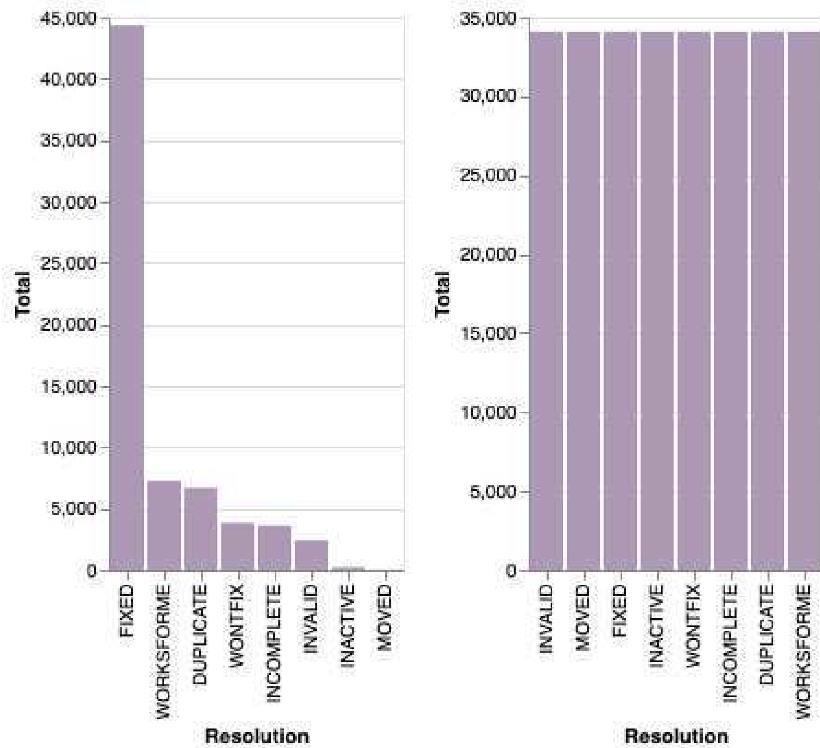
The BRs that compose the dataset were subject to data cleaning by removing field reports containing too many empty or null values as flags with approx. 86% of null data and the reports that came with an empty resolution, which is the target variable, left us

with a dataset of approximately 66K BRs. In addition, we removed features directly related to the bug's resolution, which, if maintained, could bias the classification results. For instance, *total_changes_status* indicates the number of changes in the status, and *total_changes_resolution* shows the number of changes in the bug's history, which is the model's target label.

3.4 Data Balancing

The data obtained from Bugzilla is naturally unbalanced, where more bugs are resolved as FIXED than any other resolution, as stated in Table 3.4. Generally, unbalanced data lead the machine learning model to bad results because the unbalanced data is complex for the model to generalize to the minimal classes [46; 57]. Due to the presence of unbalanced data, we conducted experiments to address this issue by employing the oversampling method [14] during the training phase. We chose this approach because it effectively increases the data for classes with fewer samples, thus ensuring a more equitable representation of all classes in the dataset. For instance, the class MOVED had insufficient data, but oversampling helped equalize the data distribution across all classes. By applying the oversampling technique, we aimed to mitigate the potential bias caused by the skewed data distribution, leading to more robust and reliable results in our evaluation.

As the last step from the data setup in Figure 3.2, we applied data balancing to the training data. In addition, this unbalance is expected as most reports tend to be system failures that must be solved. While Figure 3.3(a) illustrates the distribution of the dataset before performing the balancing method, Figure 3.3(b) shows the distribution of the (training) data balanced by RandomOverSampler.



(a) All data before balancing. (b) Training data after balancing.

Figure 3.3: Bug report distribution according to resolution.

3.4.1 Oversampling

The oversampling method basically generates new samples for the minimal classes. In order to perform the data balancing, we used RandomOverSampler from Imbalanced Learn [49], an open-source library. Studies have shown that imbalanced data can lead to poor performance of machine learning models, especially when the minority class (in this case, bug reports resolved as MOVED or INACTIVE) is of interest. In such cases, oversampling techniques showed to be effective in improving the performance of machine learning models by balancing the distribution of classes [23; 44]. For instance, a study by Chawla et al. [23] found that oversampling techniques, such as SMOTE, improved the performance of machine learning models on imbalanced datasets. Another study by He and Garcia (2009) showed that oversampling techniques improved the performance of a Support Vector Machine (SVM) classifier on imbalanced datasets [44]. Therefore, using oversampling techniques in our study is justified to improve the performance of machine learning models in

predicting the resolution of bug reports.

3.5 Machine Learning Algorithms

In this research, we employ five machine learning algorithms to predict the resolution of bug reports. The selection of these algorithms is justified based on their established effectiveness in classification tasks and their suitability for bug report resolution prediction, as described in the Background section 2. Each model brings unique characteristics and advantages contributing to the comprehensive analysis of bug report data. In Figure 3.4, we show the adopted protocol for applying the five ML algorithms on the balanced BR dataset for classifying their resolution status.

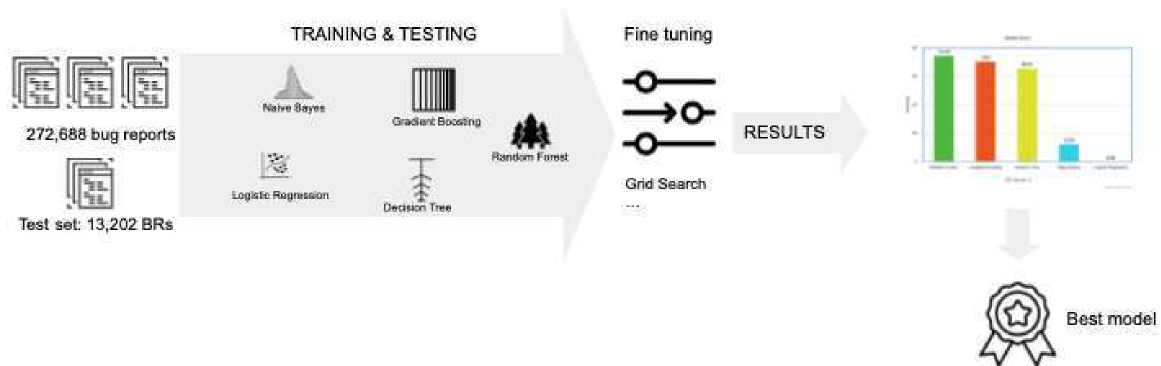


Figure 3.4: Pipeline for machine learning algorithms training, testing, and fine-tuning.

Once the dataset has been processed, including adding dummy features, we proceed to split it into training and testing sets, and finally, we balance the training set. We divided the dataset into 80% for training and 20% for testing [40]. We then apply five different ML models to classify the final resolution of bug reports: Random Forest, Gradient Boosting, Logistic Regression, Decision Tree, and Gaussian Naive Bayes. To achieve the best results with the ML algorithms, we experiment with various combinations of procedures mentioned earlier. We create different scenarios by trying out different combinations, as shown in Table 3.5. Following the pipeline depicted in Figure 3.4, we perform fine-tuning on all five models to identify the optimal set of parameters that yield the best classification model. Finally, the best model is selected to predict the BR according to its resolutions.

3.6 Experimental Procedure

We experimented the ML algorithms using features dummyfication [56] on the dataset, which transforms the non-numeric features into numeric ones; data normalization [82], basically normalizing the range of independent variables; data balancing using the RandomOversample [14] method due to the considerable variance in the bug report resolutions; we split the dataset into training and testing (80% and 20%); and, finally experimented five machine learning algorithms to classify the bug report final resolution, which are:

- Random Forest, which is a meta-estimator that fits several decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control overfitting;
- Gradient Boosting builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions;
- Logistic Regression is usually used for cases where the dependent variable is binary and multiclass, which allows estimating the probability of the variable occurring according to a given event, in the context of NLP and classification of a word impacting the classification of a sentence and in our context is the impact of the features on classifying the bug report resolution;
- Decision Tree is a non-parametric supervised learning method used for classification and regression;
- Gaussian Naive Bayes is a probabilistic classifier based on Bayes' theorem that assumes independence between terms.

We decided to experiment with these models because they are well fit for the multi-class classification tasks, have different characteristics, and bring different insights for our study as detailed in Table 2.1, as well these models could be easily used in future research to continue this work. The primary purpose of this work is to understand the features or actions that help increase the quality of bug reports and the chances of developers fixing them. In order to achieve the best results for the machine learning algorithms used, we applied combinations of different procedures from those mentioned before, forming the scenarios in Table 3.5.

Besides executing the models in the data without modification, which we named Original, we also explored the scenarios where we just balanced the data (B), generating equal distribution through the BR resolutions; normalized the data (N), which transforms the values of numeric features to a common scale, without distorting the differences in their ranges; also applied the dummification (D), which is the process of converting categorical features into numerical features by creating dummy variables (indicates the presence or absence of a category); dummification and normalization together(DN); applied feature dummification and data balancing (DB), and also combined these two with normalization (DNB).

Table 3.5: Data processing scenarios.

Scenario	Normalization	Balancing	Dummification
Original			
N	✓		
B		✓	
D			✓
NB	✓	✓	
ND	✓		✓
BD		✓	✓
NBD	✓	✓	✓

After executing the five models in the aforementioned scenarios, we fine-tuned the best one to increase the metrics values, improve the models' quality, and assess the results. We also selected the model with the higher F1 measure, initially grouped the classification, evaluated the results, and conducted a more profound analysis. In Chapter 4, we detail the results.

3.7 Evaluation Metrics

The classification models will classify the bug report in one of the eight possible resolutions FIXED, DUPLICATE, INVALID, INCOMPLETE, WONTFIX, MOVED, WORKSFORME, and INACTIVE. We will evaluate the model results considering the well-known metric accuracy, precision, recall, and F1-measure [73]. These metrics are commonly used

to evaluate the performance of multi-class classification models and are used to choose the best machine learning model, answering the research questions according to the best model selection involving fine-tuning and feature importance. Concerning analyzing the numeric feature, we check their correlation by applying the method of Pearson [26], Kendall [60], and Spearmen [85]. Also, we list the features that are most important to best model and analyze them.

Accuracy 3.1 is a metric that measures the overall correctness of the model's predictions across all classes. In other words, it measures how many bug reports were correctly classified into their respective categories by the model.

$$Accuracy = \frac{CorrectPredictions}{TotalPredictions} = \frac{TruePositives + TrueNegatives}{TotalInstances} \tag{3.1}$$

Precision 3.2 is a metric that measures the proportion of correctly predicted positive instances out of the total instances predicted as positive for a particular class. In other words, it measures how many of the bug reports that were classified into a certain category by the model were actually in that category.

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives} \tag{3.2}$$

Recall 3.3, also known as sensitivity or true positive rate, measures the proportion of correctly predicted positive instances out of the total actual positive instances for a particular class. In other words, it measures how many of the bug reports that were actually in a certain category were correctly classified into that category by the model.

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives} \tag{3.3}$$

The F1 Measure 3.4 is a harmonic mean of precision and recall. It provides a single metric that balances the trade-off between precision and recall.

$$F1Measure = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{3.4}$$

In addition, we use histograms to understand the data distribution and boxplot to evaluate the machine learning models classifications trying to understand the relations between some features and how they influence the results. The boxplot analysis considers the total observations for each classification, the mean, standard deviation, minimum, maximum, and first, second, and third quartile.

Chapter 4

Results and Discussion

This chapter presents the outcomes and analysis derived from evaluating bug report resolution prediction using different machine learning models. We discuss the findings in light of the research questions stated in this study, which aim to evaluate the performance of the models, understand the characteristics of the best classification model, and identify the bug report features strongly linked to bug resolution.

The first research question (RQ1) sought to determine which machine learning model performs best in predicting bug resolution status. To address this question, we evaluated five prominent machine learning models: Random Forest, Decision Tree, Gradient Boosting, Logistic Regression, and Gaussian Naive Bayes. We assessed the models using various metrics, including accuracy, precision, recall, and F1 score, to measure their predictive capabilities in determining bug resolution status. Additionally, we conducted a sensitivity analysis to examine how parameter variations affected the model's performance, providing a further understanding of its strengths and weaknesses and performing different data processing and grouping strategies.

The second research question (RQ2) aims to discern the key BR fields in determining its resolution, as elucidated by the best machine learning model. Addressing this inquiry necessitates ranking the most influential features employed by the optimal model. Subsequently, we conducted an in-depth analysis of the pertinent data to discern how these identified features provide critical insights into the information a reporter should emphasize while formulating and completing a bug report. By undertaking this comprehensive investigation, we sought to unravel the intrinsic connections between the prominent BR fields and their

consequential impact on the resolution process.

In this section, we will present and discuss the results obtained from addressing these research questions. The performance of each machine learning model will be compared, highlighting the model that outperformed others in predicting bug resolution status. The characteristics and features that contributed significantly to the performance of the best classification model will be explored, clearing the factors behind its success. Moreover, the bug report features strongly linked to bug resolution status will be discussed, emphasizing their importance in improving the quality and effectiveness of bug reports.

4.1 RQ1: What is the best machine learning model for bug report resolution classification?

The execution of the aforementioned five ML models within the seven scenarios described in Table 3.5 resulted in the outcomes presented in Table 4.1. According to Table 4.1, the best performing model overall is the Random Forest, with an F1 measure of 72.32% in the scenario where all features have (BD) applied, followed by Gradient Boosting in the same scenario. Gaussian Naive Bayes model has the lowest F1 measures across all scenarios, indicating that it might not perform as well as other models in this specific context. Regarding the impact of preprocessing sets, it seems that applying (BD) yields better F1 measures than subsets of preprocessing. The performance of the Logistic Regression model is quite sensitive to the choice of preprocessing, with high variability in F1 measures across different scenarios. It performs well (55.21%) only when applying (N). On the other hand, Random Forest and Gradient Boosting models are relatively stable across different scenarios, with consistently high F1 measures. Overall, Table 4.2 shows that Random Forest outperformed the other models, achieving an accuracy of 74.46%, precision of 71.81%, and F1 score of 72.32%, despite Random Forest being the best model, the metrics demonstrate that there is still room for improvement.

Table 4.3 presents the classification metrics for each class individually, as determined by the Random Forest model. The model demonstrated excellent performance in classifying the FIXED resolution, achieving an F1-score of 96%. The model also classified The INCOMPLETE class as a relatively high degree of precision, with an F1-score of 56%. However, the

Table 4.1: F-Measure by scenario.

ML Model	Original	N	B	D	NB	ND	BD	NBD
Gaussian Naive Bayes	53.43%	1.45%	17.12%	54.09%	2.88%	4.45%	15.78%	4.86%
Logistic Regression	50.35%	55.21%	5.00%	50.55%	43.89%	66.81%	0.00%	58.21%
Decision Tree	63.23%	63.75%	63.30%	68.00%	63.26%	67.66%	66.66%	65.73%
Random Forest	68.60%	68.35%	69.71%	71.32%	69.85%	70.92%	**72.32%**	72.19%
Gradient Boosting	67.65%	67.44%	65.37%	71.07%	65.81%	70.60%	69.22%	68.70%

Table 4.2: Best models metrics results.

Model	Accuracy	Precision	Recall	F1 Score
Gaussian Naive Bayes	57.34%	56.26%	57.34%	54.09%
Logistic Regression	71.28%	65.51%	71.28%	66.81%
Decision Tree	68.19%	67.83%	68.19%	68.0%
Random Forest	74.46%	71.81%	74.46%	72.32%
Gradient Boosting	73.57%	71.02%	73.57%	71.07%

model encountered difficulties in accurately predicting the remaining classes. For example, the MOVED class had only seven bug reports in the test sample, and the model was unable to classify any of them correctly. This suggests that further model refinement may be necessary to improve its performance in classifying these classes.

4.1.1 Models' Fine-tuning

In order to achieve optimal performance and enhance the predictive capabilities of the machine learning models, we conducted a process of fine-tuning. Fine-tuning involves adjusting the hyperparameters and parameters of the models to improve their ability to learn patterns and make accurate predictions [74; 67; 9; 24; 65]. This section outlines the fine-tuning process employed for each one of the five machine learning models used in this research. In the subsequent discussion, we elucidate the hyperparameters employed in fine-tuning each ML algorithm, underscoring their significance in the process. We provide succinct explanations to outline the nature and purpose of these hyperparameters, along with the specific values

Table 4.3: Random Forest metrics for individual classes.

Class	Precision	Recall	F1-score	Total
FIXED	86%	96%	90%	8488
INCOMPLETE	68%	47%	56%	712
WONTFIX	50%	30%	38%	728
DUPLICATE	40%	36%	38%	1310
WORKSFORME	45%	45%	45%	1463
INVALID	52%	17%	26%	452
INACTIVE	50%	2%	5%	42
MOVED	0%	0%	0%	7

chosen for their implementation.

1. Random Forest and Decision Tree: For the Random Forest model, the following hyperparameters were fine-tuned:

   - **max_depth**: This hyperparameter determines the maximum depth of a decision tree. It restricts the number of levels in the tree from the root to the leaf nodes. A deeper tree can capture more complex relationships in the data but also increase the overfitting risk. The *max_depth* hyperparameter accepts various values, including:
     - *None*: If set to *None*, the tree is grown until all the leaves are pure (i.e., all samples in a leaf belong to the same class) or until all leaves contain less than the *min_samples_split* samples required for further splitting. Essentially, there is no constraint on the maximum depth, and the tree can become very deep. We chose None due our due the large dataset with ample samples and features, as it can help the model extract as much information as possible from the data.
     - *15*, *35*, and *50*: These specific values indicate that the maximum depth of the tree is limited to *15*, *35*, and *50* levels, respectively. By restricting the maximum depth, the model becomes less complex, reducing the risk of overfitting. These values were chosen for experimenting, limiting the model to shallow threes and deeper three for handling overfitting and capturing more complex patterns.

- **max_leaf_nodes**: This hyperparameter sets the maximum number of leaf nodes that a decision tree can have. It controls the growth of the tree by restricting the splitting process. The *max_leaf_nodes* hyperparameter accepts different values, including:
  - *None*: When set to None, there is no restriction on the number of leaf nodes, and the tree continues to split until all nodes are pure or contain fewer samples than the *min_samples_split* requirement. We chose None in order to give the tree full freedom to partition the data as it sees fit.
  - *250*, *500*, *750*, *1000*, and *5000*: These specific values indicate the maximum number of leaf nodes allowed in the tree. The tree is pruned earlier by setting a smaller value, resulting in a simpler model with fewer leaf nodes. We chose smaller values like 250 or 500 to encourage the tree to stop splitting earlier, creating simpler models with fewer leaf nodes. This can help combat overfitting, especially when dealing with smaller datasets or when computational resources are limited. And the higher values we chose to make it possible for the three to grow higher.

2. Gradient Boosting: Fine-tuning of the Gradient Boosting model involved adjusting the following hyperparameters:

   - **n_estimators**: This hyperparameter represents the number of weak learners (decision trees) that are sequentially added to the ensemble. It controls the overall complexity and capacity of the model. Higher values of *n_estimators* can potentially improve the model's performance by allowing more iterations, but they also increase computational time and memory requirements. Therefore, the following values experimented with are *50* and *100*. The values were chosen in order to explore the trade-off between model performance and computational cost.
   - **max_depth**: This hyperparameter determines the maximum depth of each decision tree in the ensemble. It limits the number of nodes or levels in a tree. A smaller *max_depth* value constrains the complexity of the trees and helps prevent overfitting. Conversely, a larger *max_depth* allows the trees to capture more intricate relationships in the data. Therefore, the following values experimented with

are *3* and *8*. We chose the values to test whether deeper trees can extract more complex patterns from the data without overfitting.

- **min_samples_split**: This hyperparameter specifies the minimum number of samples required to split an internal node during building a decision tree. It controls the trade-off between increasing model complexity and preventing overfitting. A smaller value of *min_samples_split* can result in more complex trees and may lead to overfitting, while a larger value promotes simpler trees. Therefore, the following values experimented with are *2* and *5*. We chose the values for examining the trade-off between tree complexity and the risk of overfitting. This helps determine the optimal minimum sample size for node splitting.

- **min_samples_leaf**: This hyperparameter sets the minimum number of samples required to be at a leaf node. It determines the minimum size of the terminal nodes of the decision trees. Similar to *min_samples_split*, a smaller value promotes more complex trees, while a larger value encourages simpler trees. Therefore, the following values experimented with are *1* and *5*. We chose the values in order to assess the impact of leaf node size on model performance. Smaller values can lead to more complex trees, while larger values encourage simpler trees with fewer terminal nodes.

- **max_features**: This hyperparameter controls the number of features to consider when looking for the best split at each node. It affects the randomness and diversity of the trees in the ensemble. A higher value or *None* means considering all features, while a smaller value restricts the number of features considered for splitting. Therefore, the following values experimented with are *None*, *1*, and *5*. Experimenting with values of None, 1, and 5 helps to understand how the diversity and randomness of feature selection impact the model's performance. Using None means considering all features, while smaller values restrict the number of features considered for splitting. This exploration helps determine the most relevant features for the task.

- **subsample**: This hyperparameter determines the number of samples to train each weak learner. It controls the randomness of the training data used for each itera-

tion. A value less than *1.0* introduces stochasticity and can help reduce overfitting by providing diversity in the training process. Therefore, the following values experimented with are *0.5* and *1*, indicating that two different experiments were conducted with 50% and 100% of the training data used, respectively. The values were chosen for experimenting with values of 0.5 and 1, so evaluating how the use of a subset of the training data affects the model's performance.

3. Logistic Regression: For Logistic Regression, the fine-tuning process focused on the regularization parameter:

   - **C**: Regularization parameter. The strength of the regularization term, either *L1* or *L2* regularization, was tuned to prevent overfitting and improve the model's generalization ability. Therefore, the following values experimented with are *0.0001*, *0.001*, *0.01*, *0.1*, *1*, *10*, *100*, *1000*, *10000*, and *100000*. We chose these values in order to explore the trade-off between model complexity and overfitting. Smaller values of C encourage sparsity in the coefficients, while larger values allow for more flexible models.
   - **multi_class**: Multi-Class. The parameter assuming the values *multinomial* is used to specify that the model should use the cross-entropy loss function and predict a multinomial probability distribution for multiclass classification problems. Chosen due to the nature of the problem to be multiclass.
   - **solver**: Solver, specifies the algorithm to be used in the optimization problem. For multiclass problems, the *lbfgs* solver is one of the options that can be used. It stands for Limited-memory Broyden-Fletcher-Goldfarb-Shanno, which is an optimization algorithm in the family of quasi-Newton methods that approximates the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm using a limited amount of computer memory. We chose it because it is efficient and often works well for logistic regression with multiclass problems.

4. Gaussian Naive Bayes: Being a probabilistic classifier, Fine-tuning of Gaussian Naive Bayes involved feature selection and preprocessing rather than hyperparameter tuning.

   - **var_smooting**: The portion of the largest variance of all features added to vari-

ances for calculation stability. However, in our experiment, we chose five random values that range from a high value of *1.00000000e+00* (i.e., no smoothing) to a low value of *1.00000000e-09* (i.e., strong smoothing). As the dataset has a large number of features with low variance, it may be appropriate to use a higher value of *var_smoothing* to avoid the problem of zero probabilities. The values used are *1.00000000e+00*, *5.62341325e-03*, *3.16227766e-05*, *1.77827941e-07*, *1.00000000e-09*. The choice of these specific values allows for an exploration of the impact of smoothing on the model's stability and performance. It helps identify the optimal level of smoothing for the specific dataset and the characteristics of its features.

Throughout the fine-tuning process, manual experimentation and automated techniques, such as grid search, were employed to explore the hyperparameter space and identify the optimal settings for each model [48]. The best results achieved are shown in Table 4.4. After performing model fine-tuning, the best Random Forest was the one with the same metric values as before – we assume that the hyperparameters chosen were already the best ones in the chosen scope. On the other hand, Decision Tree had its metric values increased, where the accuracy went from 61.19% to 72.45% and F1 from 68% to 70.71%. However, the Gradient Boosting, Naive Bayes, and Logistic regression matrix did not change significantly.

Table 4.4: Model results after fine-tuning.

Model	Accuracy	Precision	Recall	F1 Score
Random Forest	74.46%	71.81%	74.46%	72.32%
Gradient Boosting	70.3%	74.3%	70.3%	72.02%
Decision Tree	72.45%	70.1%	72.45%	70.71%
Gaussian Naive Bayes	64.97%	42.21%	64.97%	51.17%
Logistic Regression	71.28%	65.51%	71.28%	66.81%

Given the confusion matrix displayed in Figure 4.1, we observe that Random Forest performed well in predicting FIXED, with a high precision (71.81%) and recall (74.96%). It correctly identified most of the actual FIXED reports, but still confusion between WORKSFORME and DUPLICATE. The confusion between DUPLICATE and INVALID could be

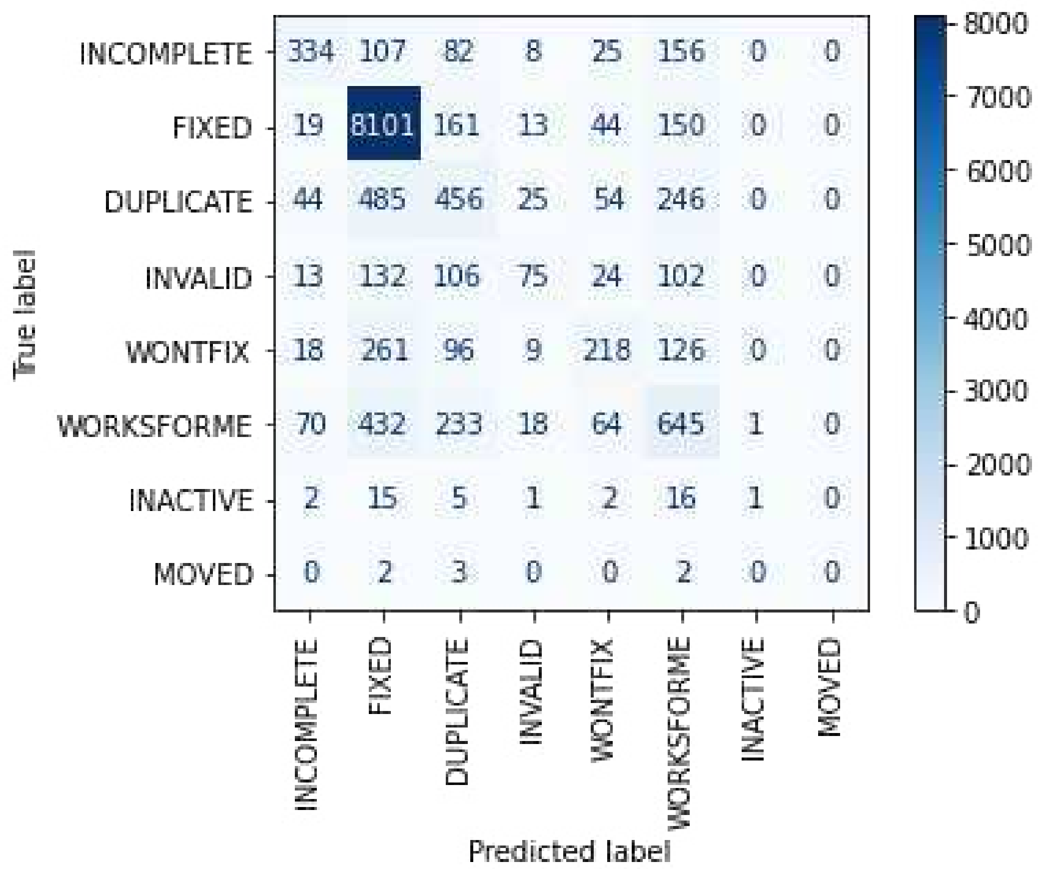


Figure 4.1: Random Forest confusion matrix.

because these two classes might share similarities in the text content of bug reports. Users might describe issues as either DUPLICATE or INVALID interchangeably, making it challenging for the model to differentiate between them. The model's confusion between INCOMPLETE and other classes may be because INCOMPLETE reports lack complete information, making them harder to classify accurately. Additionally, the terminology used in INCOMPLETE reports may overlap with other categories. Similar to WONTFIX, distinguishing WORKSFORME reports from other classes might be challenging because they could contain vague or ambiguous descriptions. Additionally, the model may need to learn subtleties in user feedback to make more accurate predictions in this category. Overall, the errors can be attributed to the complexity of the classification task, potential ambiguity in bug report descriptions, and the need for more informative features or refined modeling techniques. Addressing these challenges may require additional data preprocessing, feature engineering, or exploring more advanced machine learning algorithms to improve classification accuracy for all classes.

4.1.2 Grouping Resolutions

To explore the effect of different class distributions, we grouped the bug reports by their resolution status into various categories. We compared these categories with the baseline case of using eight classes named "Original". These categories could be useful for different purposes. For example, some users might only care about whether a bug report will be FIXED or not, while others might want to distinguish between FIXED, INVALID, and IGNORED bug reports. We also examined how the performance of the best models changed with different numbers of classes in the dataset. We used Random Forest to classify the new groups as it is the best model from the five we experimented with.

When grouping them into two classes, we specifically distinguished the FIXED resolution from the rest, which we labeled as NOT FIXED. This distinction is illustrated in Table 4.5. The overall metric values showed an approximate 14% increase compared to the original classification before grouping. Notably, the model achieved the highest values, around 90%, when predicting the FIXED bug reports. For the NOT FIXED classification, the model achieved values ranging from 83% to 86%. Guo et al. [42] work in predicting which bugs get FIXED in Microsoft Windows got a precision of 68% and recall of 64%. Hence, our

classifier outperformed his, but we consider that his work evaluated a private source and used a Logistic Regression model. In contrast, we have used an open-source and Random Forest model, which might be the reason for this difference.

Table 4.5: Metrics for Random Forest when grouping the resolutions into two classes.

	accuracy	precision	recall	f1-score	support
Overall	88.84%	88.95%	88.84%	88.88%	13202
FIXED	90%	92%	90%	91%	8488
NOT FIXED	86%	83%	86%	85%	4714

Table 4.6 presents the results obtained from grouping the data into three classes: FIXED, IGNORED (consisting of MOVED, WONTFIX, and INACTIVE), and INVALID (comprising INVALID, INCOMPLETE, WORKSFORME, and DUPLICATED). At Yuanrui work [35], the class INVALID has the same set of resolutions as ours. The overall metric values generally hover around 84%, which is an improvement compared to the Original model with eight classes that achieved approximately 10% less precision. Despite the grouping, FIXED remains the class with the highest prediction accuracy. However, IGNORED exhibits lower values, with an F1-score of 28%. Our approach outperformed Yuanrui's on classifying INVALID by a difference of approximately 11%.

Table 4.6: Metrics for Random Forest grouping the resolutions into three classes.

	accuracy	precision	recall	f1-score	support
Overall	84.67%	84.09%	84.67%	83.54%	13202
FIXED	92%	91%	92%	91%	8488
IGNORED	18%	63%	18%	28%	777
INVALID	83%	74%	83%	78%	3937

In our analysis, we further subdivided the dataset into four distinct classes. The FIXED class represents bug reports that were successfully resolved. Bug reports marked as WONTFIX or DUPLICATED we reclassified as IGNORED. Similarly, we grouped bug reports classified as INACTIVE or MOVED under the INACTIVE class, while those classified as INCOMPLETE, WORKSFORME, or INVALID we grouped under the INCOMPLETE class.

Table 4.7: Metrics for Random Forest grouping the resolutions into four classes.

	accuracy	precision	recall	f1-score	support
Overall	78.87%	77.36%	78.87%	77.8%	13202
FIXED	94%	88%	94%	91%	8488
IGNORED	39%	53%	39%	45%	2038
INACTIVE	2%	50%	2%	4%	49
INCOMPLETE	64%	63%	64%	63%	2627

This classification scheme allows for a more nuanced analysis of the bug reports, enabling us to distinguish between those that were fixed, those that were ignored, those that are no longer active, and those that lack complete information.

The overall metric values for this classification approach are approximately 78%, which remains considerably higher compared to the classification rate for the Original classes (4% higher). The class with the highest prediction accuracy is still FIXED, followed by INCOMPLETE, with a prediction rate of 63%. However, INACTIVE still has much to improve, possibly due to the few samples for the classes that compose it. For detailed results, refer to Table 4.7.

Table 4.8 summarizes the results obtained from our analysis, comparing the model's overall performance when using different grouping schemes against the original classification. As observed, the model's performance improves as the number of classes decreases, which is consistent with our expectation that fewer classes would enable the model to generalize its classifications better. In all cases, including the original classification, the model demonstrated high accuracy in predicting the FIXED category, consistently achieving metrics of 90% or higher. Overall, we experimented with three different grouping schemes for the original classes in the dataset, each yielded satisfactory results and can be applied in different contexts as appropriate.

4.1.3 Cross-Validation

Cross-validation is a technique used to evaluate the performance of a machine learning model [7]. We used to assess how well the model will generalize to an independent data set. In this

Table 4.8: Comparison of classification groupings.

No. of Classes	Accuracy	Precision	Recall	F1-Score
8 (Original)	74.46%	71.81%	74.46%	72.32%
4	78.87%	77.36%	78.87%	77.8%
3	84.67%	84.67%	84.67%	83.54%
2	88.84%	88.95%	88.84%	88.88%

context, we performed cross-validation to support the results of the best model obtained from the experiment. The cross-validation metrics are: Average Accuracy: 75.11%, Average Precision: 72.40%, Average Recall: 75.11%, and Average F1 Score: 72.89%. The k-fold used was 10. These results suggest that the Random Forest model fits the data well, but there is still room for improvement. The cross-validation results support the initial model metrics obtained from the experiment. The average accuracy, precision, recall, and F1 score are all consistent with the initial metrics obtained from the experiment. This indicates that the model is not overfitting to the training data and is generalizing well to new data.

4.1.4 Results

Given the provided accuracy of 74.46%, precision of 71.81%, recall of 74.46%, and F1-score of 72.32% for Random Forest in classifying bug report resolutions within the categories FIXED, INVALID, INCOMPLETE, WONTFIX, WORKSFORME, MOVED, DUPLICATED, and INACTIVE, along with the balanced dataset and fine-tuning applied, we can draw several insights regarding the model's performance and its implications in the context of bug reporting and software engineering processes.

Precision and recall are particularly significant in bug report resolution classification. High precision, such as the 85% precision achieved for the FIXED category, minimizes false positives, ensuring that identified issues are genuinely in need of resolution. This is vital in preventing resources from being wasted on incorrect bug reports. In contrast, a lower precision would lead to more false positives, potentially overwhelming the development team with non-essential bug reports. In practical terms, a high precision is often more desirable than a high recall in bug reporting because it ensures that the identified issues are genuine,

reducing the risk of wasting resources on false positives. The specific results for classifying the FIXED category with 85% precision and 95% recall highlight the model's ability to accurately identify bug reports that should be resolved as FIXED while capturing the majority of relevant issues.

The accuracy of 74.46% indicates the overall correctness of the model's predictions across all classes. It represents the proportion of correctly classified instances out of the total number of instances. This value suggests that the model achieves a relatively high level of accuracy in predicting bug report resolutions. However, it is important to note that accuracy alone might not provide a complete understanding of the model's performance, as the class distribution can influence it in the dataset. A higher accuracy indicates that the model is making more correct predictions overall, which can support decision-making processes in bug fixing and software development.

In the context of bug report resolution classification, precision and recall play crucial roles in evaluating the performance of machine learning models. In summary, precision focuses on minimizing false positives, ensuring that identified issues are genuinely in need of resolution. It emphasizes the accuracy of positive predictions. Recall, on the other hand, prioritizes capturing all actual positive instances, emphasizing the model's ability to find genuine issues, even if it leads to some false positives. In this context, precision is of paramount importance.

A high precision value, such as the 85% precision achieved by the Random Forest model in classifying the resolution category FIXED, signifies that when the model predicts a bug report as belonging to the FIXED category, it is highly likely to be correct. This is critical in bug reporting because it minimizes the occurrence of false positives, ensuring that resources are not wasted on incorrectly identified issues. In contrast, a low precision would lead to a high number of false positives, potentially overwhelming the development team with non-essential bug reports.

Considering the results for the best classification, in the context of bug reporting and software engineering, the specific results obtained by the Random Forest model for classifying the resolution category FIXED (precision of 85%, recall of 95%, and F1-score of 90%) indicate the following findings:

- Precision of 85%: This implies that out of all bug reports predicted as belonging to the

FIXED resolution category, approximately 85% of them are correctly classified. In bug reporting, this high precision suggests that when the model identifies a bug report as FIXED, it is likely to be accurate, minimizing false positives. This precision value indicates that the model's predictions can be relied upon to a significant extent when identifying bug reports that should be resolved as FIXED.

- Recall of 95%: The recall value of 95% indicates that the model successfully captures approximately 95% of all bug reports that should have been classified as FIXED. This high recall value implies that the model is effective at identifying the majority of bug reports that require a FIXED resolution. It minimizes false negatives, ensuring that a significant proportion of relevant FIXED issues are identified.

- F1-score of 90%: The F1-score combines precision and recall into a single metric, providing an overall assessment of the model's performance. The F1-score of 90% indicates a good balance between precision and recall for classifying FIXED bug reports. This suggests that the model achieves high accuracy in identifying and categorizing bug reports that should be resolved as FIXED. The F1-score considers false positives and negatives, providing a reliable measure of the model's effectiveness.

Fine-tuning machine learning models: Fine-tuning machine learning models can potentially lead to improved results. However, due to resource constraints, we had to limit the scope of our hyperparameter exploration by choosing a smaller set. Although we could have achieved better outcomes with a broader range of hyperparameters, we chose to explore a smaller set. We found that the F1 Score had a range between 66.81% and 72.32%, and the best models were Gradient Boosting and Random Forest with very close metrics. We chose Random Forest as it showed to be the best F1 Score (72.32%) and was relatively stable across different scenarios with consistently high F1 Scores. Overall, the fine-tuning process did not substantially improve the models' performance. Random Forest remained the best model for predicting bug report resolution, particularly for the FIXED class, with metric values consistently around 90%. These results remained unchanged even after fine-tuning.

Grouping bug report resolution: In the context of grouping bug report resolutions, Table 4.8 illustrates that as the number of classes decreases, the results improve. Grouping the resolution classes enhances the model's performance, with Random Forest proving to be the

most effective in predicting whether a bug report will be resolved as FIXED or not (approx. 88% accuracy and 89% precision). However, when it is necessary to discern the specific resolution class within the eight classes, the model achieves an overall accuracy of approximately 74%. Notably, when the bug report is classified as FIXED, the accuracy increases to 90%. With a precision of 88.95% on classifying whether the bug report will be fixed or not, it brings up a very good machine learning model that can be more precise in telling if the bug will be successfully fixed.

These findings are valuable in software engineering processes as they provide a more accurate and efficient way to identify and prioritize bug reports for resolution. By leveraging the model's predictions, software developers and teams can focus their efforts on addressing the identified FIXED issues, improving software quality, and enhancing user satisfaction. The high precision value contributes to a more effective bug-triaging process, facilitating timely bug fixes and ultimately leading to a more robust and reliable software system.

4.2 RQ2: Which bug report features are more strongly linked to the bug resolution status?

In this section, we aim to address research question RQ2, which focuses on identifying the BR (Bug Report) features that appear to have a relationship with bug report resolution. Understanding the relationship between specific BR features and the resolution status can provide valuable insights into the factors influencing bug fixing and help improve bug management processes.

We conducted a thorough analysis of bug report features to examine their impact on the prediction of bug resolution, particularly for the FIXED category. Figure 4.2 illustrates the ten influential features significantly affecting the model's prediction. Among these features, three are the bug report ID, which relates to the timing the bug was filed as it is sequential; text-related: summary, description, and comments, also including the number of comments overall and made by the reporter; changes related: number of changes, number of changes made by the reporter; and the users and reporter engagement: number of different users commenting; number of different users doing changes in the bug report; and finally the number of attachments added in the comment section and the bug report severity. The feature impor-

tance, also known as variable importance, provides insights into the most relevant features according to the Random Forest model. It plays a crucial role in understanding the problem at hand and can potentially guide model improvements through feature selection techniques. The feature importance values range from 0 to 1, and their sum is normalized to 1. Following the Scikit Learn API, the Random Forest's built-in *feature_importances_* is calculated as the mean and standard deviation of the accumulated impurity decrease within each tree. As follows, we discuss more about these features.

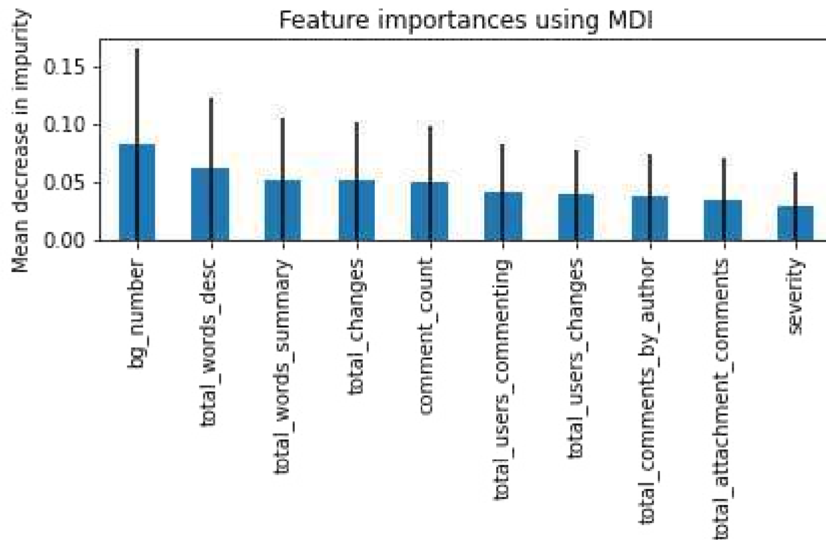


Figure 4.2: Random Forest features importance.

Due to time constraints, our evaluation focused exclusively on the most important features of the best model, leaving the assessment of the remaining four models pending. As a potential avenue for future research, it would be valuable to explore how these other models corroborate the significance of these features or potentially identify alternative features as the most important. It's important to note that our analysis may not directly align with developers' opinions but rather emphasizes the factors that aided the model in its final resolution classification, pinpointing feature importance. Our hypothesis centers on studying the features associated with the "FIXED" resolution category, with the aim of discerning which aspects of the reports classified as "FIXED" played a pivotal role in assisting developers in resolving the reported bugs.

Bug report ID (bg_number): The fact that the bug report ID appeared as the most important feature caught our attention, prompting us to delve deeper into the analysis to un-

derstand the underlying reasons. Upon further investigation, we discovered that the Scikit Learn API imposes a limitation on the method used to calculate feature importance. Specifically, features with a large number of unique values are assigned higher importance. Given that the bug report ID consists of unique values, it is likely the primary reason why this feature obtained the top position in the importance ranking. We chose to retain the bug report IDs despite their sequential numbering, as this semantic feature, wherein higher numbers correspond to more recent bugs, still holds valuable information. This decision preserves important knowledge within our dataset.

Number of words in description (total_words_desc): represents the count of words in the bug report description. Figure 4.4 showcases a notable distinction in the number of words between bug reports accurately predicted by the Random Forest model and those incorrectly predicted. For instance, consider bug 1276465[1], which has a description containing 723 words, and the model correctly classifies it as FIXED. Figure 4.3 is a snippet from the description of this bug report. To further investigate its impact on the model's decisions, we conducted an analysis focusing on the number of words in the bug report description.

We have manually observed that bug reports with a high number of words in the description (above 400) often include stack traces/backtraces/logs for the reported bugs. Likewise, Schroter et al. [80], the stack traces are very helpful in fixing bug reports, highlighting that up to 60% of FIXED bug reports that contain stack traces involved changes to one of the stack frames. They also show that the average lifetime of these reports is significantly lower than those with no stack traces.

Figure 4.4 displays a boxplot illustrating the distribution of values for *total_words_desc* across the dataset. The boxplot is categorized into three groups: "right" represents bug reports that were correctly labeled and predicted as FIXED, "wrong" represents bug reports labeled as FIXED but predicted as another resolution status, and "Others" encompasses all other cases such as other than FIXED resolutions. By examining this boxplot, we can gain insights into how the number of words in the bug report description influences the model's predictions. In the boxplot depicted in Figure 4.4, we have chosen a limit of 500 words for the bug report description to ensure better visibility of the boxes. However, it is worth noting that the actual maximum number of words differs among the categories. For the "Others"

[1]Bugzilla link: `https://bugzilla-dev.allizom.org/show_bug.cgi?id=1276465`

Assignee
Description • 7 years ago

```
There is no impact on the tool itself, everything works fine, but the jsconsole is cluttered with error messages.

STR:
1. Open `data:text/html,<p>hello` url
2. Open the devtools
3. Go to the style editor tab
4. Click on the "New" button in the left panel

Expected Result:
No error in the jsconsole

Actual Result:
There is an error in the jsconsole

Full error :
Full Message: TypeError: autocompleteMap.get(...) is undefined
Full Stack: getInfoAt@resource://gre/modules/commonjs/toolkit/loader.js -> resource://devtools/client/sourceeditor
/autocomplete.js:384:19
StyleSheetEditor.prototype._highlightSelectorAt<@resource://devtools/client/styleeditor
/StyleSheetEditor.jsm:587:16
TaskImpl.prototype._run@resource://gre/modules/commonjs/toolkit/loader.js -> resource://devtools/shared
/task.js:310:39
TaskImpl@resource://gre/modules/commonjs/toolkit/loader.js -> resource://devtools/shared/task.js:272:3
createAsyncFunction/asyncFunction@resource://gre/modules/commonjs/toolkit/loader.js -> resource://devtools/shared
/task.js:246:14
StyleSheetEditor.prototype._onMouseMove/this.mouseMoveTimeout<@resource://devtools/client/styleeditor
/StyleSheetEditor.jsm:574:7
setTimeout handler*StyleSheetEditor.prototype._onMouseMove@resource://devtools/client/styleeditor
/StyleSheetEditor.jsm:573:29
EventListener.handleEvent*StyleSheetEditor.prototype.load/<@resource://devtools/client/styleeditor
/StyleSheetEditor.jsm:464:9
Handler.prototype.process@resource://gre/modules/Promise.jsm -> resource://gre/modules/Promise-backend.js:937:23
this.PromiseWalker.walkerLoop@resource://gre/modules/Promise.jsm -> resource://gre/modules/Promise-
backend.js:816:7
Promise*this.PromiseWalker.scheduleWalkerLoop@resource://gre/modules/Promise.jsm -> resource://gre/modules
/Promise-backend.js:747:11
this.PromiseWalker.schedulePromise@resource://gre/modules/Promise.jsm -> resource://gre/modules/Promise-
backend.js:779:7
this.PromiseWalker.completePromise@resource://gre/modules/Promise.jsm -> resource://gre/modules/Promise-
backend.js:714:7
Editor.prototype.appendTo/onLoad@resource://gre/modules/commonjs/toolkit/loader.js -> resource://devtools/client
```

Figure 4.3: Bug report with a stack trace (cropped).

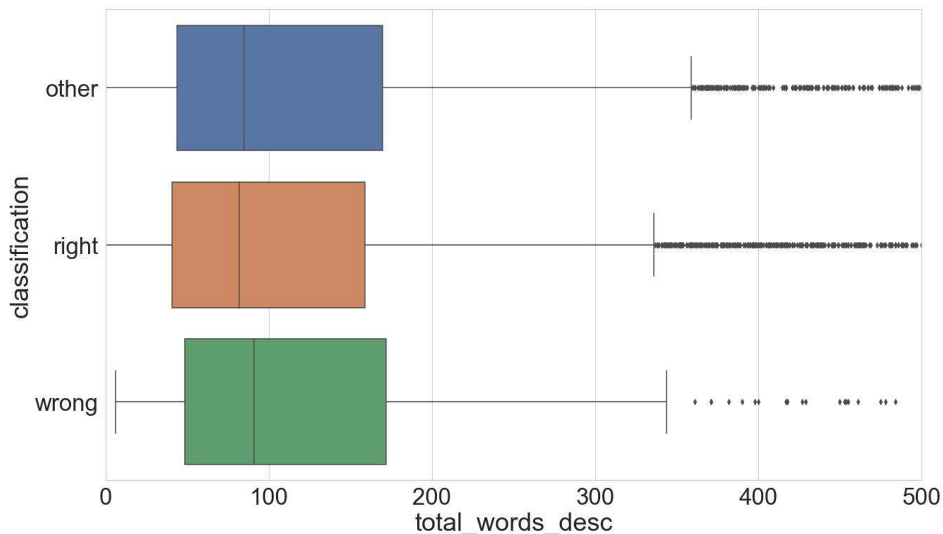


Figure 4.4: Boxplot for the Feature *total_words_desc*.

Table 4.9: Metrics number of attachments in the comments for bug report description under 40 words.

Classification	Count	Mean	STD	Min	25%	50%	75%	Max
Overall	13202	165.9	331.6	0	43	83	163	9857
wrong	83	0.1	0.34	0	0	0	0	2
right	1931	5.2	12.44	0	1	2	5	361

category, the maximum number of words reaches 7286, while for the "Right" category, it is 9857, and for the "Wrong" category, it is 3991.

Examining the mean number of words in the description across all categories, we observe a similarity, ranging from 162 to 171 words, which is in close proximity to the overall average of approximately 165 words. Furthermore, the quartile values for all categories exhibit similar patterns. However, the notable distinction lies in the presence of a longer tail in the "Right" category compared to the "Wrong" category, indicating a higher number of outliers. This implies that bug reports correctly classified as FIXED by the models tend to exhibit more extreme values in terms of word count in the description, in contrast to those erroneously classified as FIXED. However, as it could seem to be, there is no evidence that as large the bug report description as it increases the chances of being resolved, but might the content of it. Naturally, the description is not the indicated place to insert any type of trace but in the attachment. As a future work, the content of the description could be evaluated in terms of quality and clarity and checked whether it is influential in the final bug report resolution.

As observed, textual fields have proven to be crucial assets in the bug reporting process. This significance aligns seamlessly with the concepts of severity and priority, as when a bug is deemed severe or of high priority, it necessitates a more comprehensive and detailed report. This emphasis on thoroughness ensures that the essential information needed for an effective bug fix is adequately provided, thereby enhancing the resolution process for critical issues.

Number of attachments in comments (total_attachment_comments): Upon conducting a more detailed analysis, we delved into bug report classifications as FIXED specifically for

cases where the description contained fewer than 40 words. Interestingly, we discovered that the presence of attachments added to the comments could significantly influence the resolution of these bug reports as FIXED by the developers. Table 4.9 provides a comparison between the "wrong" and "right" classes showing the statistics since the mean, standard deviation, and quartiles, where we examined the presence of attachments in the comments. Notably, the occurrence of attachments in the comments is more prevalent in the "right" class than the "wrong" class. This observation leads us to speculate that the attachments added to the comments potentially contribute additional information to the bug report, compensating for any potential lack of detail in the short description. Hence, it appears that including attachments in the comments may serve as a compensatory factor, supplementing the limited information in bug reports with shorter descriptions and aiding in their accurate classification as FIXED by the models.

Overall, recognizing the significance of adding attachments in the comments section, particularly for bug reports with shorter descriptions, we analyzed the feature "total_attachment_comments". This feature represents the number of attachments added to the bug report comments. We generated a boxplot visualization to gain insights into this feature, as depicted in Figure 4.5. What can be difficult to follow is how clear are the description and how it could potentially influence the total of attachments. The common knowledge is that if a bug report is well described, there is no need to add further information as comments or attachments. However, we haven't evaluated the textual information from the comments section nor the attachment's relevance. What we can draw from the influence of attachments in the bug report is that the description was not enough to precisely inform the bug faced.

Upon examining the boxplot, it becomes evident that the "right" class exhibits a substantially higher number of attachments added in the comments than the other classes. The average number of attachments in the comments across all classes hovers around three. However, the "wrong" class shows an average of less than one attachment, whereas the "right" class boasts an average of nearly five attachments. In this context, the mere presence of comments alone does not wield significant influence; however, the inclusion of multiple attachments enhances the likelihood of a bug report being successfully classified as FIXED. Further delving into the bug report resolutions, we sought to understand how the number of attachments was distributed across different resolution categories. Table 4.10 presents descriptive statis-

tics specifically pertaining to the number of attachments found in the comments section for each resolution category. It is remarkable to observe that the FIXED resolution stands out prominently, featuring a higher attachment prevalence than any other resolution. In fact, the third quartile value for the "FIXED" resolution indicates an average of approximately five attachments, while the other resolutions hover around one attachment.

This analysis underscores the importance of attachments in the comments section, particularly in bug reports with shorter descriptions. The findings reveal that the "right" class, which accurately predicts bug reports as FIXED, tends to possess more attachments in the comments section, potentially contributing to a more comprehensive and informative bug report. The number of attachments in the comments section of a bug report appears to impact the resolution, particularly in cases where the resolution is classified as FIXED. This observation suggests a higher prevalence of attachments in bug reports that are ultimately resolved.

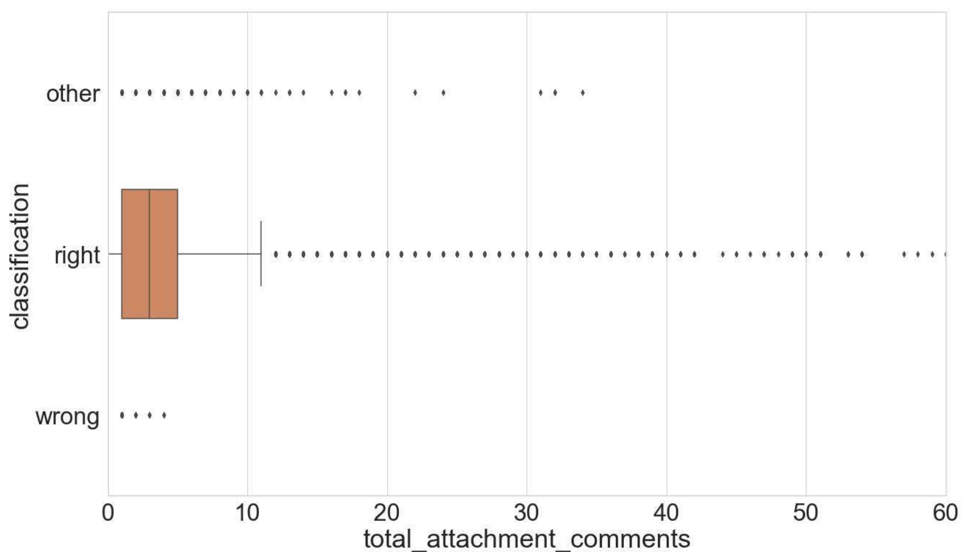


Figure 4.5: Boxplot for the Feature *total_attachment_comments*.

Number of comments added by the author to the bug report (total_comments_by_author): This feature indicates the interaction by the author in the bug that he or she has reported. We conducted a correlation analysis involving the feature *total_attachment_comments*, using the Pearson correlation coefficient. Among the various features examined, the one demonstrating the highest positive correlation with *total_attachment_comments* was *total_comments_by_author*. The correlation coefficient was approximately 0.7, indicating a

Table 4.10: Descriptive statistics of attachments in the comments for bug report description for all resolutions.

Label	Count	Mean	STD	Min	25%	50%	75%	Max
DUPLICATE	6559	0.5	2.4	0	0	0	0	137
FIXED	42574	4.6	10.7	0	1	2	5	1045
INACTIVE	219	1.6	9	0	0	0	1	105
INCOMPLETE	3499	0.3	1.8	0	0	0	0	58
INVALID	2323	0.4	1.6	0	0	0	0	32
MOVED	45	0.8	1.5	0	0	0	1	6
WONTFIX	3688	0.7	2.73	0	0	0	0	62
WORKSFORME	7101	0.5	1.6	0	0	0	0	36

strong positive correlation between the two features. This finding suggests that as the number of attachments in the comments section increases, there is a corresponding increase in the number of comments contributed by the bug report author. It is reasonable to assume that the author has actively participated by including attachments within the comments. However, further analysis is required to gain deeper insights into the specific content and context of the comments provided by the authors.

This correlation analysis highlights the potential relationship between the presence of attachments in the comments section and the level of engagement and interaction by the bug report author. It signifies the need for a more detailed examination of the comments contributed by authors to fully understand the impact of their involvement and the role of attachments in the resolution process.

Number of comments on the bug report: In order to understand the role of comments in bug reports, we conducted an evaluation based on bug report resolution. We examined how comments are distributed among eight different resolutions and how the classification model utilizes them. Table 4.11 presents each resolution's medians and means of comment distribution. Additionally, it includes columns indicating the values associated with the right and wrong classifications made by the Random Forest model:

- **Wrong Median**: Median value for instances with wrong classification;

- **General Median**: Median value for all instances;
- **Right Median**: Median value for instances with right classification;
- **Wrong Mean**: Mean value for instances with wrong classification;
- **General Mean**: Mean value for all instances;
- **Right Mean**: Mean value for instances with right classification.

Table 4.11: Bug report resolution and comments statistics.

Label	**Median**			**Mean**		
	Wrong	**General**	**Right**	**Wrong**	**General**	**Right**
DUPLICATE	6.0	5.0	4.0	9.5	7.9	4.9
FIXED	5.0	12.0	12.0	7.9	17.7	18.1
INACTIVE	9.0	8.5	4.0	17.9	17.5	4.0
INCOMPLETE	6.0	5.0	3.0	12.5	10.9	9.1
INVALID	5.0	4.0	2.0	8.5	7.6	3.2
MOVED	10.0	10.0	NaN	11.0	11.0	NaN
WONTFIX	6.0	5.0	4.0	12.0	10.1	5.7
WORKSFORME	7.0	6.0	5.0	12.8	12.2	11.5

Analyzing the table results, it becomes apparent that the median comment counts for wrong and right classifications vary across bug report resolutions. This suggests that the number of comments in a bug report may influence classification accuracy. For instance, the INACTIVE resolution tends to have a higher median comment count for wrong classifications than for right ones. Furthermore, the mean comment counts also exhibit variation between wrong and right classifications for bug report resolutions. This indicates that the average number of comments in a report can impact classification accuracy. For example, the mean comment count for the right classification of the WORKSFORME resolution is higher than the mean comment count for the wrong classification. It could indicate the user interaction in the comment section discussing that they could not reproduce the reported bug. In order to understand and ensure this, a detailed investigation in the comment section is needed.

There is a noticeable difference in comment counts between wrong and right classifications. Wrong classifications tend to have lower median and mean comment counts than the right ones. This implies that bug reports with a higher number of comments may have a higher likelihood of being correctly classified. It is important to note that these conclusions pertain specifically to the relationship between comment counts and the accuracy of bug report resolution classification within the context of the Random Forest classification algorithm. Other factors, such as comment quality, the bug's specific nature, and the classification algorithm's choice, may also influence the classification results, which will be investigated in future work.

Table 4.12: Statistics of the number of changes by bug report resolution.

Label	**Median**			**Mean**		
	Wrong	**General**	**Right**	**Wrong**	**General**	**Right**
DUPLICATE	6.0	5.0	4.0	8.5	7.2	4.8
FIXED	5.0	12.0	12.5	6.2	15.1	15.5
INACTIVE	7.0	7.0	3.0	9.3	9.1	3.0
INCOMPLETE	6.0	4.0	3.0	7.6	5.9	3.9
INVALID	5.0	5.0	3.0	6.9	6.4	3.8
MOVED	6.0	6.0	NaN	8.8	8.8	NaN
WONTFIX	7.0	6.0	5.0	8.8	8.0	6.2
WORKSFORME	7.0	6.0	5.0	9.3	8.0	6.3

Number of changes in the bug report (total_changes): Refer to the count of modifications made to the report over time. In a given context where a bug report has been active, meaning it has attracted attention and ongoing discussions, a higher number of changes often could imply a more complex or critical issue. This increased level of activity can lead to a greater likelihood of the bug report being eventually resolved as FIXED [91] as we can see in Table 4.12, the media and mean for the FIXED bug reports are the highest compared to the other resolution. The reason behind this correlation could be that active bug reports tend to receive more attention from developers and stakeholders, facilitating a deeper understanding of the problem and promoting more comprehensive solutions. Also, these results are aligned with the feature *total_user_changes* that indicates the changes made by the reporter, which for the

FIXED resolution is the highest.

Given the critical role of the *priority* feature in bug report resolution, it is intriguing that the model does not assign it significant importance. To delve into the underlying reasons, we conducted a comprehensive investigation. Surprisingly, we discovered that approximately 20% of the bug reports in our dataset lack a priority designation, marked as '–'. Consequently, we hypothesize that the model may not have recognized the significance of this feature due to its absence in a substantial portion of the dataset. Surprisingly, the Random Forest analysis ranked severity as the tenth most important feature, which was unexpected considering the critical role of severity and its semantic significance in bug reports. To gain a deeper insight into this anomaly, further investigation is warranted to understand the distribution of severity within the dataset and the factors influencing its importance.

However, these fields need to be thoroughly discussed with developers to ascertain, based on their expertise, the extent to which they are crucial in comprehending the bug. This exploration aims to confirm whether these fields are vital solely for the machine learning model or if they indeed reflect the practical information required at the moment a developer works to address the reported bug.

To conclude and answer RQ2, our analysis of the features identified as most important by Random Forest indicates that the majority are related to textual information, including the summary, description, and comments of bug reports. Additionally, some other features are connected to updates in bug reports regarding changes made during the resolution process. We observed that bug reports with more detailed descriptions tend to be resolved as FIXED, and in many cases, they also contain stack trace information. This highlights the importance of adding attachments and extra information to bug reports, a point discussed in other works as well. Furthermore, attachments have proven significant in resolving bugs as FIXED, especially for reports with limited descriptions, suggesting that they provide complementary information. The presence of comments and changes signals that the bug report has been active and continuously updated, indicating a relationship between these updates and the resolution of the bug.

4.3 Research Implications

During the development of this work, we acquired valuable insights into composing bug reports, drawing from studies rooted in our research and informed by the outcomes of the models we experimented with. As a result, we have put forth a set of guidelines that can be instrumental in enhancing bug report composition. Such guidelines, like to the one presented here, could serve as a means to introduce best practices for novice bug reporters, thereby optimizing their contributions. Additionally, for reporters who are integral members of a software development team, these guidelines can aid in comprehending the significance of the fields outlined below.

- *Summary*: Should be succinct while providing a summary that clearly articulates the core problem.
- *Description*: While our research indicates that reports with more detailed descriptions tend to result in faster resolutions, we emphasize that content quality is more important than sheer word count. The description should include:
  - *Steps to Reproduce*: A step-by-step sequence that outlines how to recreate the bug.
  - *Actual Result*: A description of the current, problematic system behavior.
  - *Expected Result*: An explanation of the normal, expected system behavior.
- *Product and Component*: Accurate specify the product and component where the bug is happening.
- *Priority and Severity*: Set these fields to appropriately ensure that the issue receives the necessary attention.
- *Attachments*: Our findings indicate that the inclusion of traces or logs significantly contributes to faster bug resolutions. Screenshots or screen recordings are especially helpful, as they provide a visual context for understanding the issue. Any attachment that complements the description, particularly those highlighting the problem's location in the code, is invaluable.

- *System Information*: Must provide precise information, it is imperative to specify the environment in which the bug occurred. This includes details such as the software version, platform, and any relevant hardware.

Effective bug reporting is crucial for the smooth resolution of software issues. A well-crafted bug report, following guidelines such as providing a succinct summary, detailed description with steps to reproduce, and attaching relevant materials like traces or screenshots, could help developers understand and address the problem. Accurate product and component identification, along with setting appropriate priority and severity, might further streamline the bug resolution process. Additionally, including precise system information helps developers replicate the issue in the same environment, ultimately expediting the resolution and enhancing the overall efficiency of the bug fixing process.

4.4 Threats to Validity

There are several potential threats to the validity of this study, which can affect the credibility and generalizability of the results. We categorized these threats into four types: construct validity, conclusion validity, external validity, and internal validity.

4.4.1 Construct Validity

While conducting research on bug report resolution prediction, we identified several threats to construct validity. Construct validity refers to the extent to which the measurements and manipulations employed in a study accurately represent the concepts being investigated. To ensure the construct validity of this work, the following threats were considered and mitigated:

- **Measurement Bias**: There is a risk of measurement bias if the selected features and metrics do not adequately capture the relevant aspects of bug report resolution. We carefully chose a comprehensive set of features to address this threat based on their potential influence on bug report resolution. The selected features, such as resolution status, number of comments, severity, priority, and attachment presence, were deemed relevant and representative of the underlying construct. By including these features,

the study aimed to encompass various dimensions of bug report resolution and minimize measurement bias.

- **Sampling Bias**: Sampling bias can undermine construct validity if the selected bug reports do not represent the broader population effectively. A large dataset of 68,492 bug reports from Bugzilla was collected to mitigate this threat. By leveraging Bugzilla as a widely used bug-tracking system, the researchers aimed to include bug reports from diverse projects, thus increasing the likelihood of capturing a representative sample. This approach helped minimize sampling bias and enhance the generalizability of the findings.

- **Choice of Machine Learning Models**: Selecting inappropriate or suboptimal machine learning models can compromise construct validity if the models do not effectively capture the relationships between the features and bug report resolution. We utilized Five machine learning models to overcome this threat (Random Forest, Gradient Boosting, Logistic Regression, Decision Tree, and Gaussian Naive Bayes). This diverse set of models was chosen based on their effectiveness in classification tasks and suitability for bug report resolution prediction. By employing multiple models, the study aimed to capture various aspects of the bug report data and increase the robustness of the findings.

By addressing these threats to construct validity, the research ensured that the measurements and manipulations used accurately represented the concepts of interest. The selection of relevant features, the use of a representative dataset, thorough data cleaning and preprocessing, and the application of appropriate machine learning models collectively contributed to maintaining the construct validity of the study. These steps enhance the reliability and validity of the research findings and support the credibility of the conclusions drawn from the analysis.

4.4.2 Internal Validity

During the research on bug report resolution prediction, several threats to internal validity were identified and effectively addressed. Internal validity pertains to the extent to which the

observed effects in a study can be confidently attributed to the manipulated variables rather than extraneous factors. To ensure the internal validity of this work, the following threats were carefully considered and mitigated:

- **History**: The occurrence of external events during the research period could influence the bug report resolution, leading to potential threats to internal validity. To address this threat, the bug report dataset was carefully selected to cover a significant time period, ideally representing a range of external events. The study aimed to minimize the impact of specific historical events on the bug report resolution prediction models by including bug reports from various projects and time frames.

- **Selection Bias**: Selection bias can threaten internal validity if the bug reports used for analysis are not representative of the broader population. A large dataset of bug reports was collected from Bugzilla, a widely used bug-tracking system to address this threat. By leveraging a diverse range of projects and including a large number of bug reports, the study aimed to minimize selection bias and increase the generalizability of the findings.

- **Instrumentation**: Changes in the measurement instruments or tools used to collect bug report data can introduce threats to internal validity. A consistent methodology and data collection process were employed throughout the study to mitigate this. The bug report dataset was obtained using standardized queries and procedures, ensuring uniformity in data collection. By maintaining consistent instrumentation, the study aimed to minimize the potential impact of measurement variations on the internal validity of the research.

In order to maintain the internal validity of the bug report resolution prediction research, we took several steps to address potential threats. We carefully selected the bug report dataset, included a diverse range of bug reports, and used consistent instrumentation throughout the study. These measures helped us establish reliable causal relationships and supported the validity of our conclusions. By addressing these threats, we ensured that the observed effects could confidently be attributed to the variables we manipulated.

4.4.3 External Validity

External validity refers to the extent to which the findings of a study can be generalized to other populations, settings, or conditions beyond the specific context of the research. In this work, several external threats to the validity of the findings can be identified. However, the following measures addressed these threats and enhanced the study's external validity.

- **Generalizability of Bug Tracking Systems**: The study focuses on bug reports collected from the Bugzilla bug tracking system, which may differ from other bug tracking systems regarding workflows, policies, and data structures. In order to mitigate this threat, a comprehensive description of the Bugzilla bug-tracking system was provided, including its key features and characteristics. This allows readers to assess the generalizability of the findings to bug-tracking systems with similar attributes. Furthermore, future research could explore other bug-tracking systems to validate the findings' applicability across different platforms.

- **Diversity of Software Projects**: The dataset used in the study consists of bug reports from various projects, including Firefox, Bugzilla, Mozilla, and Thunderbird. The specific nature and characteristics of these projects might influence the findings. However, to address this threat, our study acknowledged the diversity of the software projects included in the dataset. By explicitly stating the projects involved, readers can assess the transferability of the findings to other software projects.

- **Data Quality and Reliability**: The accuracy and completeness of the bug reports and associated information in Bugzilla may introduce external threats to the study's validity. To mitigate this threat, we peformed rigorous data cleaning and validation processes. The study also provided details on the data cleaning steps undertaken to ensure the reliability of the dataset.

- **Limitations of Bugzilla API**: The study relies on the Bugzilla API to collect the bug report data. Any limitations or constraints in the API's functionality or availability may impact the completeness or accuracy of the dataset. Although to address this threat, our study acknowledged the reliance on the Bugzilla API and its potential limitations. By providing transparency about the data collection process, readers can evaluate the

potential impact of API-related issues on the findings. Also, the API is very much used in the process of bug extraction and is already solid in the community.

By acknowledging these external threats and implementing the aforementioned mitigation strategies, this work aimed to enhance the study's external validity.

4.4.4 Conclusion Validity

Conclusion validity refers to the degree to which the conclusions drawn from the data are accurate and supported by evidence. This study potentially threatens conclusion validity due to the limited scope of our analysis. We only examined bug reports from a single open-source project, and the results may not be generalizable to other projects or types of software.

To address the potential threat to conclusion validity stemming from the limited scope of analysis, it is essential to acknowledge the need for caution when generalizing the findings of this study. While the results provide valuable insights into bug reports within the specific open-source project examined, it is crucial to recognize each software project's uniqueness and specific characteristics. Different projects may have varying development methodologies, team dynamics, and bug reporting practices, which can influence the nature and resolution of bugs.

However, using only open-source projects on Bugzilla for the analysis contributes to the validation of generalizability in several ways. Firstly, the transparency and accessibility of open-source projects' bug repositories, such as Bugzilla, allow researchers to access a vast amount of bug reports with detailed information. This transparency ensures that the findings are based on real-world data and can be scrutinized by the community, enhancing the credibility and generalizability of the conclusions drawn. Additionally, open-source projects encompass a diverse range of domains, sizes, and complexities. By including multiple open-source projects in the analysis, researchers can capture a wide variety of bug reporting practices, development methodologies, and team dynamics. This diversity helps to ensure that the findings reflect the broader software development landscape and can apply to a wider range of projects.

Chapter 5

Related Work

There are many works related to bug reports and how to identify ways to improve them to help developers solve bugs faster. Bettenburg et al. [99] surveyed 156 developers and bug reporters from open-source projects to investigate the fields expected in a bug report. The survey results showed that there are 16 important fields to fix bugs (product, hardware, observed behavior, screenshots, component, operating system, expected behavior, code examples, version, summary, steps to reproduce, error reports, severity, build information, stack traces, and test cases), and from these features, we use in our research: product, component, version, severity, summary, operating system, and attachments that could include screenshots, error reports, stack traces, and test cases. Also, they developed a tool named CUEZILLA, which evaluates bug reports and suggests the fields to be filled to improve the report and help developers fix bug reports. This tool was developed using supervised learning models, and to validate, another survey was conducted asking developers to evaluate 289 randomly chosen bug reports evaluating the quality on a five-point Likert scale. Once the bug report was evaluated, the work compared the quality feedback provided by CUEZLLA with the one provided on the survey. In conclusion, the models used on the tool achieved 45% accuracy in measuring the quality of bug reports compared to the answers on the survey. Likely our study, there is a line of looking for bug reporting improvement; their work has surveyed developers while we have extracted information from the reports, trained a ML model, and investigated which are the essential information pointed out from these models that most contribute to bug fix.

The study by Jian et al. [88] examined bug reports for an application server over four years and listed why these were invalid to reveal the weakness and mistakes in the invalid

bug report. So they have found that in addition to errors in testing, misunderstandings on functionality and environments, lack of background knowledge, problems in external systems and tools, and other reasons can lead to invalid bug reports. Xiaoxue et al. [94] also investigated invalid bug reports to understand software aging signals by analyzing the performance issues in these systems. As results were found that around 50% of the performance bug reports (PBRs) in invalid bug reports (IBRs) are related to software aging; components that undertake major tasks are more prone to aging problems; more than 50% aging-related bug reports (ARBs) lead to timeout, 33% ARBs are caused by improper control of memory or threats, and 29% ARBs are caused by inappropriate management of file operation or disk usage; hard to reproduce is the major reason that ARBs are usually closed as invalid because many aging-related bugs would temporarily disappear by restarting the system.

Yuanrui et al. [35] proposed an approach to identify if a new bug report will be valid, and it used 33 features from bug reports, then grouped along five dimensions and used a classifier to determine whether the bug is valid or not. The difference from our work is that we need to differentiate the resolution classes, and we include MOVED and INACTIVE resolutions. Also, we run another four machine learning models besides Random Forest and compare their results. Only description and summary were used by Yuanrui from the features we used. The dataset used form them includes Eclipse, Netbeans, Mozilla, Firefox and Thunderbird, while we do not include Eclipse and Netbeans in ours. Their approach achieved an F1-score for valid bug reports and F1-score for invalid ones of 74% and 67%, respectively. However, when we grouped our dataset into VALID(only FIXED BR) and NOT VALID(the other BR resolutions), the F1-score was 91% and 85%, respectively.

Davies et al. [29] research was developed to investigate how users report bugs in systems, more specifically, what information is provided, how frequently, and which are the consequences of. In the study, they examined four open-source projects (Eclipse [33], Firefox [36], Apache HTTP [6], and Facebook API [34]) for the quality and quantity of information provided in 1600 bug reports from those projects. The features studied are Observed behavior, Expected behavior, steps to reproduce, error reports, stack traces, screenshots, code examples, test cases, build information, and application code. Most of these features are extracted from the description, which we have used only in a manner to understand the length of it in terms of words, as well the features related to attachments, which they have used

for some specific type like screenshots. The research shows a clear mismatch between what developers find essential in the bug report and what reporters provide. Furthermore, 12% of the features on the bug report are provided after the first submission, making the developer spend time collecting the needed features.

Also, Anvik et al. [3] have investigated open-source projects and found that 6% of bug reports in the Eclipse project and 11% of bug reports in the Firefox project were resolved as invalid. Bachmann et al. [8] studied open-source and closed-source projects reporting that the proportion of invalid bugs together with non-reproducing ones was as high as 36%. Yuanrui et al. [35] have already worked on classifiers (Random Forest and Support Vector Machine) to identify valid and invalid bug reports. Still, in their work, they have set all the resolutions besides FIXED to be invalid bug reports and only the ones resolved as fixed to be valid bug reports. The features used were the report experience, collaboration network, completeness, readability, and text. Their results are an F1-score for valid bug reports and an F1-score for invalid ones of 74% and 67%, respectively. The features in common with our work were summary, description, attachments, and description. We propose to investigate the bug report field relationship with its resolution to bring the users' awareness of what is essential to inform when filling it. Some difference between the Yuanrui and our study is that we investigated eight different resolutions, applied machine learning models to predict them, and performed an analysis of how the features explain the model classification and how this information could help the users to understand the essential features to focus while reporting the bug. Besides, our best model was Random Forest, and when we grouped in two resolutions, INVALID and NOT VALID, we got an F1-score of 91% and 95%.

The research from Karim [59] investigates the essential information required when reporting a High Impact Bug (HIB) in software development. The study analyzes HIB reports in the Apache Camel project through qualitative and quantitative analysis. The findings reveal that four types of features (Steps to Reproduce, Stack Traces, Test Cases, and Code Examples) are frequently requested by developers when fixing HIBs, and the inclusion of requested additional information significantly impacts bug fixing time. The research aims to understand the characteristics of effective HIB reports and provide insights for improving bug reporting guidelines and tool development. Further, Karim et al. [58] extend their works aiming to improve the bug-fixing process by identifying key features that reporters

frequently miss in their initial bug report submissions and that developers require for fixing bugs. To achieve this goal, the author conducts exploratory and empirical studies on five large open-source projects from Apache and Mozilla ecosystems. They find that the additional features that reporters most often omit from their initial bug report submissions are Steps to Reproduce, Test Case, Code Example, Stack Trace, and Expected Behavior. His work differs from ours due to the bug report type we consider besides HIB, and the variety of features we have used, including textual features related to the summary, description, and comments. Additionally, we have incorporated features that pertain to the change history of the bug report. Our investigation analyzes the main characteristics of FIXED bug reports compared to seven other resolutions. Furthermore, we have employed five machine learning (Logistic Regression, Random Forest, Decision Tree, Naive Bayes, and Gradient Boosting) algorithms in our study to predict the bur report resolution and access their results. As a result, Random Forest outperforms the other with an F1-score of 72%. At the same time, they have four machine learning models (Naive Bayes, Naive Bayes Multinomial, k-Nearest Neighbors, and Support Vector Machine); the best model was Naive Bayes Multinomial with a F1-score variating between 65% to 76% across the projects.

Ding et al. [31] focuses on predicting highly impactful bugs (HIBs) in software systems by analyzing test smell occurrence in bug reports. The study utilizes a combination of test smell detection and bug report data to construct a dataset for analysis. Ding proposed a model (Random Forest) that outperforms existing baseline models in predicting HIBs within a project and achieves high mean F-Measure values over time. The model also shows promising results in cross-project prediction. The paper identifies specific test smells, such as assertion roulette (AR), complex test logic (CTL), and coupling of implementation (CI), as strong predictors of HIBs. Case studies further validate the connection between these test smells and actual bug occurrences. The proposed machine learning model was Random Forest and outperformed the baseline by 29.30%. The findings suggest that developers should focus on improving test code quality, particularly by addressing code maintainability and avoiding specific test smells, to reduce the occurrence of HIBs. The bug report features used are the same as Karim [58], and the difference from our work is in the general nature of the features, mainly the combination of HIB and test smell detection. While our work focuses on the bug report resolution FIXED in order to understand the features that influence this resolution

considering all bugs besides HIB.

Also, Zhou [98] investigates the management of bug-fixing processes in software development by analyzing the different severity and its changes. The study explores the distributions and evolutions of bug reporters and owners for high, medium, and low-severity bugs on desktop and Android platforms. The features used include time to fix the bug, title, description, description length, number of comments, number of words in the comments, priority, bug report ID, reporter, and developer experience. We have also used the bug title, description length, number of comments, priority, and bug report ID from these features. The findings highlight the significant contribution of high-severity bug owners due to resource requirements while also revealing differences in bug trends and severity classifications between platforms. Additionally, the paper applies topic analysis to extract bug topics using Latent Dirichlet Allocation (LDA), uncovering distinctions in bug characteristics across severity classes. The research provides valuable insights into bug-fixing management, resource allocation, and the dynamics between bug reporters and bug owners, emphasizing the importance of considering severity levels and topic analysis in software development processes. In our work, severity is a feature used in the models' classification. Still, the focus is not on severity itself but on the main characteristics of the FIXED bug reports and how they lead to be so.

The proposed approach from Imran [50], Bug-AutoQ, addresses the issue of incomplete bug reports by automatically generating relevant follow-up questions. It leverages a large collection of previously posted follow-up questions on GitHub to identify the most appropriate questions for a given bug report. Bug-AutoQ selects bug reports from active GitHub repositories with high bug reporting activity, focusing on longer-running projects and recently active projects. It excludes feature requests and chooses issues with concise follow-up questions in the comments, retrieving answers from comments or edits to the original bug report. The system determines the utility of a question based on the quality of its answers(according to a dataset manually annotated), considering Observable Behavior (OB), Expected Behavior (EB), and Steps to Reproduce (S2R). Bug-AutoQ demonstrates effectiveness, outperforming baselines with a Precision@1 score of 0.49 and receiving positive feedback from software developers in a survey. It successfully selects appropriate questions by evaluating their utility based on the quality of answers, prioritizing questions with more OB, EB, or S2R information. The system's performance surpasses baseline methods, achieving a Precision score

of 49%, indicating that nearly half of the recommended follow-up questions are considered valid. Feedback from software developers in a survey further supports Bug-AutoQ's effectiveness, as they found the selected follow-up questions to be useful and specific. They are also requested new information yet to be present in the bug report. The utility and compatibility functions of Bug-AutoQ play crucial roles in its performance, demonstrating its value in enhancing the bug reporting process. This work relates to ours in the line of improving the bug reporting process, mainly by discussing the content of the bug report itself.

In Guo's study [42], the authors present a descriptive and predictive statistical model to understand the factors that influence the successful fixing of software bugs in Windows Vista and Windows 7. They utilize a logistic regression model and an Analysis of Deviance test to examine the independent effects of various factors on bug fixes. The descriptive model provides insights into the correlations between factors such as bug opener reputation, severity level, and the relationship between bug opener and assignee. The coefficients in the model offer intuitive meanings, allowing comparisons between factors and their impacts on bug-fix probability. Categorical factors, such as bug source, reveal variations in bug-fix rates, while numerical and boolean factors demonstrate positive or negative correlations with bug fixes. To address the limitations of the descriptive model in predicting bug-fix probability for newly-opened bugs, the authors develop a predictive model using only factors available at the time of bug report creation. This model achieves comparable performance to the descriptive model, indicating the effectiveness of factors from the bug's initial stages. The authors evaluate the predictive model through cross-validation and by predicting bug fixes in an entirely new dataset.

Overall, they achieve satisfactory precision and recall values, 68% and 64%, respectively, when predicting Windows 7 bug fixes using the Logistic Regression model, suggesting the model's potential to prioritize bug reports during triage and aiding in the decision to close bugs selectively. The study highlights the importance of understanding the factors that influence bug fixes for efficient resource allocation and bug monitoring in software development projects, which also aligns with our interests in the sense of providing essential information in a report, resources will be efficiently used in the whole bug fixing process. The features used in common with the ones used in our work are severity, assignee, component, and resolution status and related to bug report edit, finding that many assignments are a characteristic

that makes the bug less likely to be fixed, as well as the high reputation of the reporter, makes more likely to be fixed. Finally, bugs handled by multiple teams and across multiple locations are less likely to get fixed.

In summary, the works above primarily concentrate on enhancing the bug reporting process. They aim to achieve this by comprehensively understanding and identifying crucial information, thereby providing valuable guidance to users of bug management platforms like Bugzilla. Furthermore, these efforts contribute positively to resource utilization, ultimately leading to a more efficient bug-fixing process. Our study further adds to this by specifically focusing on bug report resolutions and their feature relationships, using machine learning algorithms for classification and analysis. Our research aims to contribute to the overall bug-fixing process and resource utilization by identifying essential information in bug reports.

Chapter 6

Conclusion

As we come to the end of this work on bug report resolution and features importance, it is essential to reflect on the key findings and contributions that have been made. Throughout this research, we have examined the effectiveness of different approaches to bug report resolution and analyzed the fields that influence bug report resolution. We have also discussed the limitations of our study and identified areas for future research. This concluding chapter summarizes this work's main findings and contributions and highlights their implications for software development practice. Additionally, we will provide some recommendations for practitioners and researchers in this field.

6.1 Limitations

Several limitations to this study should be considered. First, the study focused solely on open-source software projects, and the findings may not be generalizable to closed-source or proprietary software projects. Additionally, the study only examined bug reports in English, which may limit the generalizability of the findings to other languages. However, in terms of generalization to private sources, there is a variety of projects and developers in the dataset used, which might generalize the results for other private projects as well.

Another limitation is that the study relied on automated data collection and analysis methods. While these methods were designed to be as accurate as possible, there is always the possibility of errors or inaccuracies in the data. In addition, automated methods may not be able to capture all of the nuances and complexities of bug reports and may miss important

contextual information. However, the scripts were properly tested in order to retrieve the most information from the reports accurately.

Another limitation is that the study only considered a limited set of features in bug reports, and other factors may be important in determining the quality and validity of bug reports. For example, the study did not examine the impact of the severity or complexity of the bug on the quality of the bug report.

Finally, the study focused on identifying the characteristics of bug reports that are associated with the reports' resolution. While this information is helpful for improving the quality of bug reports, it does not address the underlying reasons why these reports are FIXED. However, there were also analyzed the other resolutions in order to understand what differs from those reports resolved as FIXED and, in this way, find out the characteristic that could further be improved when filling a bug report.

6.2 Contributions

This work contributes to the bug reporting process and software engineering, specifically from the perspective of bug reporters who often face time constraints when submitting bug reports. We address two research questions to shed light on bug resolution and identify influential bug report (BR) features, focusing on enabling bug reporters to provide concise and impactful information. The contributions of this study are summarized as follows:

Empirical Investigation: This study provides empirical evidence regarding the factors that influence bug report resolution in open-source projects. By analyzing a large dataset, we have identified fields that significantly impact the resolution, thereby enriching the understanding of bug resolution dynamics.

Identification of Textual Features Linked to Bug Resolution: We uncover the strong correlation between these features and the bug resolution by analyzing BR features, particularly those related to text, such as summary, description, and comments. This finding is particularly valuable for bug reporters who face time limitations and need to prioritize the most critical information in their reports. By understanding which textual features significantly impact the bug resolution process, bug reporters can provide focused and concise reports that effectively guide developers in identifying and fixing bugs. In this way, we

found out that descriptions with many words are more likely to be resolved as FIXED. Still, when there is a large description, there is likely an added stack trace, so it could be an indication that logs are fundamental for the bug report and could be added as attachments to have cleaner textual information in the description. We can also assume that the description content could be more important than its size. In this way, our study suggests that any stack trace or log should be beforehand added to the bug report, minimizing the need to provide larger descriptions.

Importance of Bug Report Features in Predicting Resolution Status: By employing the five machine learning algorithms, Random Forest was the best, and we assessed the importance of various bug report features in predicting the resolution status. Our analysis highlights that the number of words in the description and the total words in the summary are among the most influential features. This insight empowers bug reporters to understand which aspects of their reports will most likely impact the resolution process. By emphasizing these key features, bug reporters can ensure that their bug reports provide developers with the necessary information to identify and address the reported issues efficiently. This aligns with other research highlighting the importance of adding steps to reproduce, expected behavior, and actual results enriching the textual bug report fields.

Impact of Attachments on Bug Report Accuracy and Resolution: Our study explores the significance of attachments on bug report accuracy and resolution, particularly for bug reports with limited description content. We discover that attachments when included in the comments section, play a substantial role in accurately classifying bug reports as FIXED. This finding is crucial for bug reporters who may lack time to provide extensive descriptions but can compensate by including attachments. Moreover, we observe a positive correlation between the number of attachments in the comments section and the level of engagement from the bug report author. This highlights the importance of attachments in fostering communication between bug reporters and developers, ultimately contributing to more effective bug resolution.

Predictive Models: The machine learning models developed in this research offer a practical approach for predicting bug report resolution. These models can help stakeholders in open-source projects make informed decisions and streamline the bug management process.

Practical Implications: The insights gained from this research have practical implica-

tions for bug management in open-source projects. By understanding the factors that affect bug resolution time, project managers and developers can prioritize their efforts and allocate resources more effectively to address critical bugs promptly.

Furthermore, this research can have significant implications for the ecosystem of bug reporting, triaging, and software development as a whole. By addressing the problem of low-quality bug reports and providing insights into essential bug report features. By considering this approach, we can anticipate the potential outcomes:

1. **Improved bug report quality**: This work could help developers and users write better, more informative reports by identifying the key factors contributing to a high-quality report. This could help to reduce the time and effort required to triage and fix bugs, ultimately leading to a more efficient and effective software development process;

2. **Reduced bug backlog**: By reducing the number of INVALID or INCOMPLETE bug reports that need to be manually triaged, this work could reduce the backlog of bugs that developers need to work through. This could lead to faster bug resolution times, fewer open bugs, and ultimately a more stable and reliable software product;

3. **Improved communication between users and developers**: This work could improve communication and collaboration between users and developers by helping users better understand the bug reporting process and the types of information that developers need to triage and fix bugs effectively. This could help to build stronger, more positive relationships between software developers and their user communities;

4. **More efficient use of development resources**: By reducing the time and effort required to triage and fix bugs, this work could improve the efficiency of software development teams. This could free up resources to work on other areas of software development, such as new feature development, performance optimization, or code refactoring;

5. **More robust and reliable software**: By improving the quality of bug reports and reducing the backlog of bugs, this work could ultimately lead to more robust and reliable software products. This could reduce the likelihood of critical bugs being missed or overlooked, resulting in more satisfied users and fewer support requests;

6. **Knowledge and Insights Generation**: Through the analysis of bug report resolutions and influential features, valuable insights can be gained regarding the bug resolution process. This knowledge can help researchers, practitioners, and stakeholders better understand the factors affecting bug resolution outcomes and inform future bug reporting and triaging practices.

In conclusion, this research provides valuable insights into the bug reporting process and software engineering by considering bug reporters' challenges in providing comprehensive information within limited time constraints. By identifying the textual features linked to bug resolution status, emphasizing the importance of key bug report features, and highlighting the impact of attachments, this study aids bug reporters in crafting concise and impactful reports. These contributions enhance the bug management process, facilitate efficient bug resolution, and support bug reporters in effectively assisting developers in finding and fixing bugs.

6.3 Future Work

As with any research study, there are always areas for improvement and avenues for further exploration. This study on bug report resolution is no exception, and there are several potential directions for future research. By identifying these areas, researchers or private initiatives can build upon the findings of this study and make even greater strides in improving the efficiency and effectiveness of bug report resolution processes. In this section, we explore some potential future works that could arise from this study.

An intriguing aspect deserving further examination is the quality of comments appended to bug reports. This investigation aims to ascertain the prevalent contexts within these comments, identify those that facilitate developers in comprehending and addressing bugs with greater efficiency, and pinpoint the types of information most frequently sought after in these comments. To validate the influence of these comments based on their quality, we could conduct interviews with developers through surveys to inquire about comment quality and how it affects bug report resolution. This may also lead to categorizing this information and understanding what reporters often miss while filling out bug reports.

A potential study could be related to the incorporation of additional features. Although

this study considered a set of commonly used features in bug report analysis, there may be other relevant information that could contribute to better prediction models. Future work could explore the inclusion of additional features such as developer and tester expertise, code complexity, historical bug report patterns, or social network characteristics to capture a more comprehensive representation of bug resolution dynamics.

There is also space to investigate advanced machine learning techniques. While the current study utilized well-established machine learning models, future research could investigate more advanced techniques to further improve predictive performance. Deep learning models, such as recurrent neural networks (RNNs) or transformers, could be explored to capture the temporal dependencies and complex relationships present in bug reports.

Evaluation of human factors could complement this work and be a line of research. Understanding the role of human factors in bug report resolution is another promising area for future investigation. There is space to explore the impact of developer experience, teamwork dynamics, or community engagement on the resolution process. By incorporating such factors, the prediction models could be augmented to provide more nuanced insights and recommendations.

Our research has primarily emphasized resolution FIXED. However, there remains ample opportunity to delve into other resolution statuses and explore their implications on the bug reporting process, drawing conclusions that can enhance the overall efficiency of bug reporting activities. It is crucial to comprehend the distinct characteristics of each resolution and devise strategies for improvement to prevent unnecessary time allocation towards bugs that may not undergo resolution. Therefore, as part of our proposed future work, we intend to comprehensively investigate various bug report resolutions beyond FIXED, enhancing our understanding of their dynamics and potential areas for enhancement.

Also, a comparison of different bug-tracking systems could be an interesting approach to understanding if the results found here extend to other systems besides Bugzilla. This study utilized Bugzilla as the bug-tracking system, but there are numerous other bug-tracking systems in use across different open-source projects. Comparing the predictive performance and characteristics of bug resolution across different bug-tracking systems could shed light on the influence of the system itself on the resolution process.

Finally, a real-time prediction and intervention is an interesting path to validate the results

and understanding if the information on which filed should be focused, provided to the user while filling out a bug report, in fact, results in more bug resolved as FIXED. Building on the predictive models developed in this study, future research could focus on developing real-time prediction systems that can provide timely recommendations to developers and project managers. Such systems could help prioritize bug reports, allocate resources efficiently, and reduce resolution time.

By exploring these future directions, we can continue to advance our understanding of bug report resolution in open-source projects, refine prediction models, and provide valuable insights and tools to support software development and bug management processes. Also, providing feedback on the important information that must be in the report so make better use of the reporters' limited time.

Bibliography

[1] Bugzilla . Bugzilla | contributing to mozilla | mozilla foundation. https://www.mozilla.org/en-US/contribute/bugzilla/. Accessed: May 12, 2023.

[2] Mozilla . Mozilla. https://www.mozilla.org. Accessed: May 12, 2023.

[3] John Anvik, Lyndon Hiew, and Gail C Murphy. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange (ETX)*, pages 35–39, 2005.

[4] John Anvik, Lyndon Hiew, and Gail C Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 361–370, 2006.

[5] John Anvik and Gail C Murphy. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20:10, 2011.

[6] Apache. Apache http. http://httpd.apache.org. Accessed: May 15, 2023.

[7] Stephen Bates, Trevor Hastie, and Robert Tibshirani. Cross-validation: What does it estimate and how well does it do it? *Journal of the American Statistical Association (JASA)*, pages 1–12, 2023.

[8] Cédric Beaulac and Jeffrey S Rosenthal. Best: A decision tree algorithm that handles missing values. *Computational Statistics*, 35(3):1001–1026, 2020.

[9] Candice Bentéjac, Anna Csörgő, and Gonzalo Martínez-Muñoz. A comparative analysis of gradient boosting algorithms. *Artificial Intelligence Review*, 54:1937–1967, 2021.

[10] Dimitris Bertsimas and Angela King. Logistic regression: From art to science. *Statistical Science*, pages 367–384, 2017.

[11] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiß, Rahul Premraj, and Thomas Zimmermann. Quality of bug reports in eclipse. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange (ETX)*, pages 21–25, 2007.

[12] Larissa Braz, Enrico Fregnan, Vivek Arora, and Alberto Bacchelli. An exploratory study on regression vulnerabilities. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering (ESEM)*, pages 12–22, 2022.

[13] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Information needs in bug reports: Improving cooperation between developers and users. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW)*, pages 301–310, 2010.

[14] Jason Brownlee. Random oversampling and undersampling for imbalanced classification. *Machine Learning Mastery*, 2020.

[15] Bugzilla. Bugzilla: Life cycle of a bug. https://www.bugzilla.org/docs/2.18/html/lifecycle.html. Accessed: May 22, 2023.

[16] Bugzilla. Bugzilla overview. https://www.bugzilla.org/docs/4.4/en/html/intro/overview.html, 2013. Accessed: May 12, 2023.

[17] Peter Bühlmann and Torsten Hothorn. Boosting algorithms: Regularization, prediction and model fitting. *Statistical Science*, 22(4):477–505, 2008.

[18] M. Hanefi CALP and Utku Köse. Planning activities in software testing process: A literature review and suggestions for future research. *preprint arXiv:1903.01222 (arXiv)*, 2019.

[19] Sergio Cerón-Figueroa, Cuauhtémoc López-Martín, and Cornelio Yá nez Márquez. Stochastic gradient boosting for predicting the maintenance effort of software-intensive systems. *Institution of Engineering and Technology (IET Software)*, 14(2):135–143, 2020.

[20] Oscar Chaparro. Improving bug reporting, duplicate detection, and localization. In *2017 IEEE/ACM International Conference on Software Engineering Companion (ICSE-C)*, pages 421–424, 2017.

[21] Oscar Chaparro, Carlos Bernal-Cárdenas, Jing Lu, Kevin Moran, Andrian Marcus, Massimiliano Di Penta, Denys Poshyvanyk, and Vincent Ng. Assessing the quality of the steps to reproduce in bug reports. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference (ESEC) Symposium on the Foundations of Software Engineering (FSE)*, pages 86–96, 2019.

[22] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. Detecting missing information in bug descriptions. In *Proceedings of the ACM Joint Meeting on Foundations of Software Engineering (FSE)*, pages 396–407, 2017.

[23] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, 2002.

[24] Fuhu Che, Qasim Zeeshan Ahmed, Fahd Ahmed Khan, and Faheem A Khan. Novel fine-tuned attribute weighted naïve bayes nlos classifier for uwb positioning. *IEEE Communications Letters*, 27(4):1130–1134, 2023.

[25] Songqiang Chen, Xiaoyuan Xie, Bangguo Yin, Yuanxiang Ji, Lin Chen, and Baowen Xu. Stay professional and efficient: Automatically generate titles for your bug reports. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 385–397, 2020.

[26] Israel Cohen, Yiteng Huang, Jingdong Chen, Jacob Benesty, Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson correlation coefficient. *Noise Reduction in Speech Processing*, pages 1–4, 2009.

[27] Bugzilla Contributors. `https://www.bugzilla.org/about/installation-list`, 2023. Accessed: June 30, 2023.

[28] Bugzilla Contributors. Bugzilla api documentation. https://bugzilla.readthedocs.io/en/latest/api/index.html, 2023. Accessed: June 30, 2023.

[29] Steven Davies and Marc Roper. What's in a bug report? In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10, 2014.

[30] Felipe Emerson de Oliveira Calixto, Franklin Ramalho, Tiago Massoni, and José Manoel Ferreira. Investigating bug report changes in bugzilla. *International Conference on Enterprise Information System (ICEIS)*, pages 55–64, 2023.

[31] Jianshu Ding, Guisheng Fan, Huiqun Yu, and Zijie Huang. Automatic identification of high impact bug report by test smells of textual similar bug reports. In *International Conference on Software Quality, Reliability and Security (QRS)*, pages 446–457, 2021.

[32] Sunil Joy Dommati, Ruchi Agrawal, Prof. Ram Mohana Reddy.G, and Sowmya Kamath. Bug classification: Feature extraction and comparison of event model using naive bayes approach. *arXiv preprint arXiv:1304.1677*, abs/1304.1677, 2013.

[33] Eclipse. Eclipse. http://www.eclipse.org. Accessed: May 15, 2023.

[34] Facebook. Facebook developers. https://developers.facebook.com. Accessed: May15, 2023.

[35] Yuanrui Fan, Xin Xia, David Lo, and Ahmed E Hassan. Chaff from the wheat: Characterizing and determining valid bug reports. *IEEE Transactions on Software Engineering (TSE)*, 46(5):495–525, 2018.

[36] Firefox. Mozilla firefox. http://www.mozilla.org/firefox. Accessed May 15, 2023.

[37] Mozilla Foundation. Mozilla wiki - toolkit. https://wiki.mozilla.org/Modules/Toolkit. Accessed: June 27, 2023.

[38] Mozilla Foundation. Thunderbird. https://www.thunderbird.net/pt-BR/. Accessed: June 27, 2023.

[39] The Eclipse Foundation. Eclipse bugzilla. `https://bugs.eclipse.org/bugs/`. Accessed: May 12, 2023.

[40] Afshin Gholamy, Vladik Kreinovich, and Olga Kosheleva. Why 70/30 or 80/20 relation between training and testing sets: A pedagogical explanation. Technical Report UTEP-CS-18-09, University of Texas at El Paso, El Paso, TX, USA, 2018.

[41] Anjali Goyal and Neetu Sardana. Performance assessment of bug fixing process in open source repositories. In *Procedia Computer Science*, volume 167, pages 2070–2079, 2020.

[42] Philip J Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows. In *International Conference on Software Engineering (ICSE)*, pages 495–504, 2010.

[43] Shubham Gupta, Shubham Gupta, Shubham Gupta, Shubham Gupta, and Shubham Gupta. Gaussian naïve bayes algorithm: A reliable technique for early detection of cancer. *Mobile Information Systems (MIS)*, 2022:2436946, 2022.

[44] Haibo He and Edwardo A Garcia. Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 21(9):1263–1284, 2009.

[45] Jianjun He, Ling Xu, Yuanrui Fan, Zhou Xu, Meng Yan, and Yan Lei. Deep learning based valid bug reports determination and explanation. In *2020 IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 184–194, 2020.

[46] Thomas Hirsch and Birgit Hofer. Root cause prediction based on bug reports. *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 171–176, 2020.

[47] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 34–43, 2007.

[48] R Hossain and Douglas Timmer. Machine learning model optimization with hyper parameter tuning approach. *Global Journal of Computer Science and Technology: D Neural Artificial Intelligence*, 21:7–13, 2021.

[49] imbalanced-learn contributors. Randomoversampler. https://imbalanced-learn.org/dev/references/generated/imblearn.over_sampling.RandomOverSampler.html. Accessed: May 12, 2023.

[50] Mia Mohammad Imran, Agnieszka Ciborowska, and Kostadin Damevski. Automatically selecting follow-up questions for deficient bug reports. In *International Conference on Mining Software Repositories (MSR)*, pages 167–178, 2021.

[51] Jiří Janák. Issue tracking systems. Diplomová práce, Masaryk University/Faculty of Informatics, 2009.

[52] Silke Janitza, Ender Celik, and Anne-Laure Boulesteix. A computationally fast variable importance test for random forests for high-dimensional data. *Advances in Data Analysis and Classification (ADAC)*, 12:885–915, 2018.

[53] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software (ESEC/FSE)*, pages 111–120, 2009.

[54] Jira. Jira. https://www.atlassian.com/software/jira. Accessed: May 12, 2023.

[55] Anvik John, Hiew Lyndon, and C Murphy Gail. Who should fix this bug? In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 361–370, 2006.

[56] Sakshi Jolly and Neha Gupta. Understanding and implementing machine learning models with dummy variables with low variance. In *International Conference on Innovative Computing and Communications (ICICC)*, volume 1, pages 477–487, 2021.

[57] Luiz Alberto Ferreira Gomes Jr., Ricardo da Silva Torres, and Mario Lúcio Côrtes. On the prediction of long-lived bugs: An analysis and comparative study using floss projects. *Information and Software Technology*, 132:106508, 2021.

[58] Md Rejaul Karim. Key features recommendation to improve bug reporting. In *International Conference on Software and System Processes (ICSSP)*, pages 1–4, 2019.

[59] Md Rejaul Karim, Akinori Ihara, Xin Yang, Hajimu Iida, and Kenichi Matsumoto. Understanding key features of high-impact bug reports. In *2017 Eighth International Workshop on Empirical Software Engineering in Practice (IWESEP)*, pages 53–58, 2017.

[60] Maurice George Kendall. *Rank Correlation Methods*. Griffin, 4 edition, 1962.

[61] Sotiris B Kotsiantis. Decision trees: A recent overview. *Artificial Intelligence Review (AIJ)*, 39:261–283, 2013.

[62] Kulamala Vinod Kumar, Priyanka Kumari, Avishikta Chatterjee, and Durga Prasad Mohapatra. Software fault prediction using random forests. In *Proceedings of 2019 Intelligent and Cloud Computing (ICICC)*, volume 1, pages 95–103, 2021.

[63] Ahmed Lamkanfi and Serge Demeyer. Predicting reassignments of bug reports-an exploratory investigation. In *2013 European Conference on Software Maintenance and Reengineering (CSMR)*, pages 327–330, 2013.

[64] Ahmed Lamkanfi, Serge Demeyer, Emanuel Giger, and Bart Goethals. Predicting the severity of a reported bug. In *2010 IEEE Working Conference on Mining Software Repositories (MSR)*, pages 1–10, 2010.

[65] Wei Li and Johannes Lederer. Tuning parameter calibration for l1-regularized logistic regression. *Journal of Statistical Planning and Inference (JSPI)*, 202:80–98, 2019.

[66] Kaiping Liu, Hee Beng Kuan Tan, and Mahinthan Chandramohan. Has this bug been reported? In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 1–4, 2012.

[67] Rafael Gomes Mantovani, Tomáš Horváth, Ricardo Cerri, Sylvio Barbon Junior, Joaquin Vanschoren, and André Carlos Ponce de Leon Ferreira de Carvalho. An empirical study on hyperparameter tuning of decision trees. *arXiv preprint arXiv:1812.02207*, abs/1812.02207, 2018.

[68] Lloyd Montgomery, Clara Lüders, and Walid Maalej. An alternative issue tracking dataset of public jira repositories. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 73–77, 2022.

[69] Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, David Lo, and Chengnian Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 70–79, 2012.

[70] Yuki Noyori, Hironori Washizaki, Yoshiaki Fukazawa, Keishi Ooshima, Hideyuki Kanuka, Shuhei Nojiri, and Ryosuke Tsuchiya. What are good discussions within bug report comments for shortening bug fixing time? In *2019 IEEE International Conference on Software Quality, Reliability, Security (QRS)*, pages 280–287, 2019.

[71] Daniel Asante Otchere, Tarek Omar Arbi Ganat, Jude Oghenerurie Ojero, Bennet Nii Tackie-Otoo, and Mohamed Yassir Taki. Application of gradient boosting regression model for the evaluation of feature selection techniques in improving reservoir characterisation predictions. *Journal of Petroleum Science and Engineering (JPSE)*, 208:109244, 2022.

[72] Anusha R Pai, Gopalkrishna Joshi, and Suraj Rane. Quality and Reliability Studies in Software Defect Management: A Literature Review. *International Journal of Quality & Reliability Management (IJQRM)*, 38(10):2007–2033, 2021.

[73] David MW Powers. Evaluation: From precision, recall and f-measure to roc, informedness, markedness and correlation. *International Journal of Machine Learning Technology (IJML)*, 2:37–63, 2011.

[74] Philipp Probst, Marvin N Wright, and Anne-Laure Boulesteix. Hyperparameters and

tuning strategies for random forest. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery (WIREs)*, 9(3):e1301, 2019.

[75] Zhao Qinghe, Xiang Wen, Huang Boyan, Wang Jong, and Fang Junlong. Optimised extreme gradient boosting model for short term electric load demand forecasting of regional grid system. *Scientific Reports*, 12(1):19282, 2022.

[76] RedHat. Bugzilla. https://bugzilla.redhat.com/. Accessed: May 12, 2023.

[77] Lior Rokach and Oded Maimon. *Decision Trees*, volume 6, pages 165–192. Spring, 2005.

[78] Mirka Saarela and Susanne Jauhiainen. Comparison of feature importance measures as explanations for classification models. *SN Applied Sciences*, 3:1–12, 2021.

[79] Iqbal H Sarker. Machine learning: Algorithms, real-world applications and research directions. *SN Computer Science*, 2(3):160, 2021.

[80] Adrian Schroter, Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. Do stack traces help developers fix bugs? In *2010 IEEE Working Conference on Mining Software Repositories (MSR)*, pages 118–121, 2010.

[81] Emad Shihab, Akinori Ihara, Yasutaka Kamei, Walid M Ibrahim, Masao Ohira, Bram Adams, Ahmed E Hassan, and Ken ichi Matsumoto. Predicting re-opened bugs: A case study on the eclipse project. In *2010 IEEE Working Conference on Reverse Engineering (WCRE)*, pages 249–258, 2010.

[82] Dalwinder Singh and Birmohan Singh. Investigating the impact of data normalization on classification performance. *Applied Soft Computing*, 97:105524, 2020.

[83] Moshe Sipper and Jason H Moore. Conservation machine learning: A case study of random forests. *Scientific Reports*, 11(1):3629, 2021.

[84] Yang Song and Oscar Chaparro. Bee: A tool for structuring and analyzing bug reports. In *Proceedings of the ACM Joint Meeting on European Software Engineering Con-*

ference (ESEC) Symposium on the Foundations of Software Engineering (FSE), pages 1551–1555, 2020.

[85] Charles Spearman. *The Proof and Measurement of Association between Two Things*, volume 100. Appleton-Century-Crofts, 1961.

[86] Klaas-Jan Stol and Brian Fitzgerald. *Guidelines for Conducting Software Engineering Research*, pages 27–62. Springer International Publishing, Cham, 2020.

[87] Abdulhamit Subasi. Chapter 3 - machine learning techniques. In Abdulhamit Subasi, editor, *Practical Machine Learning for Data Analysis Using Python*, pages 91–202. Academic Press, 2020.

[88] Jian Sun. Why are bug reports invalid? In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 407–410, 2011.

[89] Youshuai Tan, Sijie Xu, Zhaowei Wang, Zhou Zhang, and Luo Xu. Bug severity prediction using question-and-answer pairs from stack overflow. *Journal of Systems and Software (JSS)*, 165:110567, 2020.

[90] GitHub Users. An open letter to GitHub from the maintainers of open source projects. https://github.com/dear-github/dear-github, 2016. Accessed: June 21, 2023.

[91] Renan G Vieira, César Lincoln C Mattos, Lincoln S Rocha, João Paulo P Gomes, and Matheus Paixão. The role of bug report evolution in reliable fixing estimation. *Empirical Software Engineering (ESE)*, 27:164, 2022.

[92] Indika Wickramasinghe and Harsha Kalutarage. Naive bayes: Applications, variations and vulnerabilities: A review of literature with code snippets for implementation. *Soft Computing (SOCO)*, 25(4):2277–2293, 2021.

[93] GNOME Wiki. Bugzilla. https://wiki.gnome.org/Apps/Bugzilla. Accessed: May 12, 2023.

[94] Xiaoxue Wu, Wei Zheng, Minchao Pu, Jie Chen, and Dejun Mu. Invalid bug reports complicate the software aging situation. *Software Quality Journal*, 28:195–220, 2020.

[95] Xin Xia, David Lo, Emad Shihab, and Xinyu Wang. Automated bug report field reassignment and refinement prediction. *IEEE Transactions on Reliability*, 65(3):1094–1113, 2015.

[96] Xin Xia, David Lo, Xinyu Wang, and Bo Zhou. Accurate developer recommendation for bug resolution. In *2013 IEEE Working Conference on Reverse Engineering (WCRE)*, pages 72–81, 2013.

[97] Xin Xia, David Lo, Ming Wen, Emad Shihab, and Bo Zhou. An empirical study of bug report field reassignment. In *2014 IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 174–183, 2014.

[98] Bo Zhou, Iulian Neamtiu, and Rajiv Gupta. Experience report: How do bug characteristics differ across severity classes: A multi-platform study. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 507–517, 2015.

[99] Thomas Zimmermann, Rahul Premraj, and Nicolas Bettenburg. What makes a good bug report? In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 308–318, 2007.